2. Graph and network optimisation

Many decision making problems can be formulated in terms of graphs and networks:

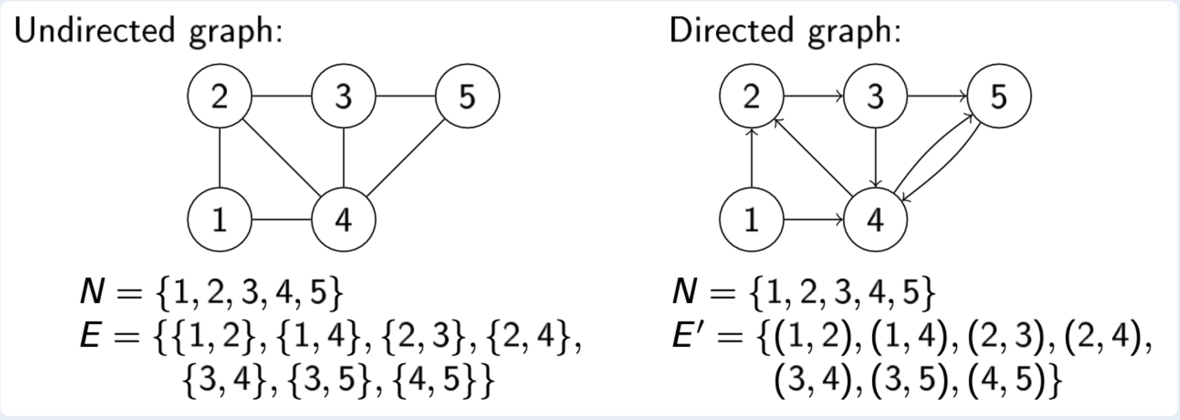
- Transportation and distribution problems
- Network design
- Location problems
- Project planning and resource management
- Timetable scheduling
- Production planning

2.1. Introduction to graphs

ⓘ Graphs

A graph is a pair $G = (N, E)$, where N is a set of **nodes** (or **vertices**) and $E \subseteq N \times N$ is a set of **pairs of nodes**.
There are two basic types of graphs:

- **Undirected**: the pairs of nodes E are not ordered
- **Directed**: the pairs of nodes E are ordered



Undirected graph:

$$N = \{1, 2, 3, 4, 5\}$$
$$E = \{\{1,2\}, \{1,4\}, \{2,3\}, \{2,4\}, \{3,4\}, \{3,5\}, \{4,5\}\}$$

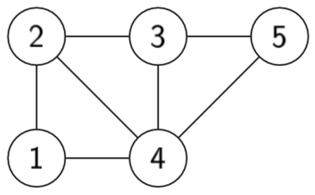
Directed graph:

$$N = \{1, 2, 3, 4, 5\}$$
$$E' = \{(1,2), (1,4), (2,3), (2,4), (3,4), (3,5), (4,5)\}$$

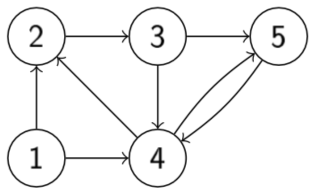
For example, a road network which connects n cities can be modelled by a graph.

Properties of graphs

- Two nodes are **adjacent** if they are connected by an edge
- An edge e is **incident** in a node v if v is an endpoint of e
- For undirected graphs, the **degree** of a node is the number of incident edges
- For directed graphs, the **in-degree** (respectively, **out-degree**) of a node is the number of arcs that have it as a head (respectively, tail):

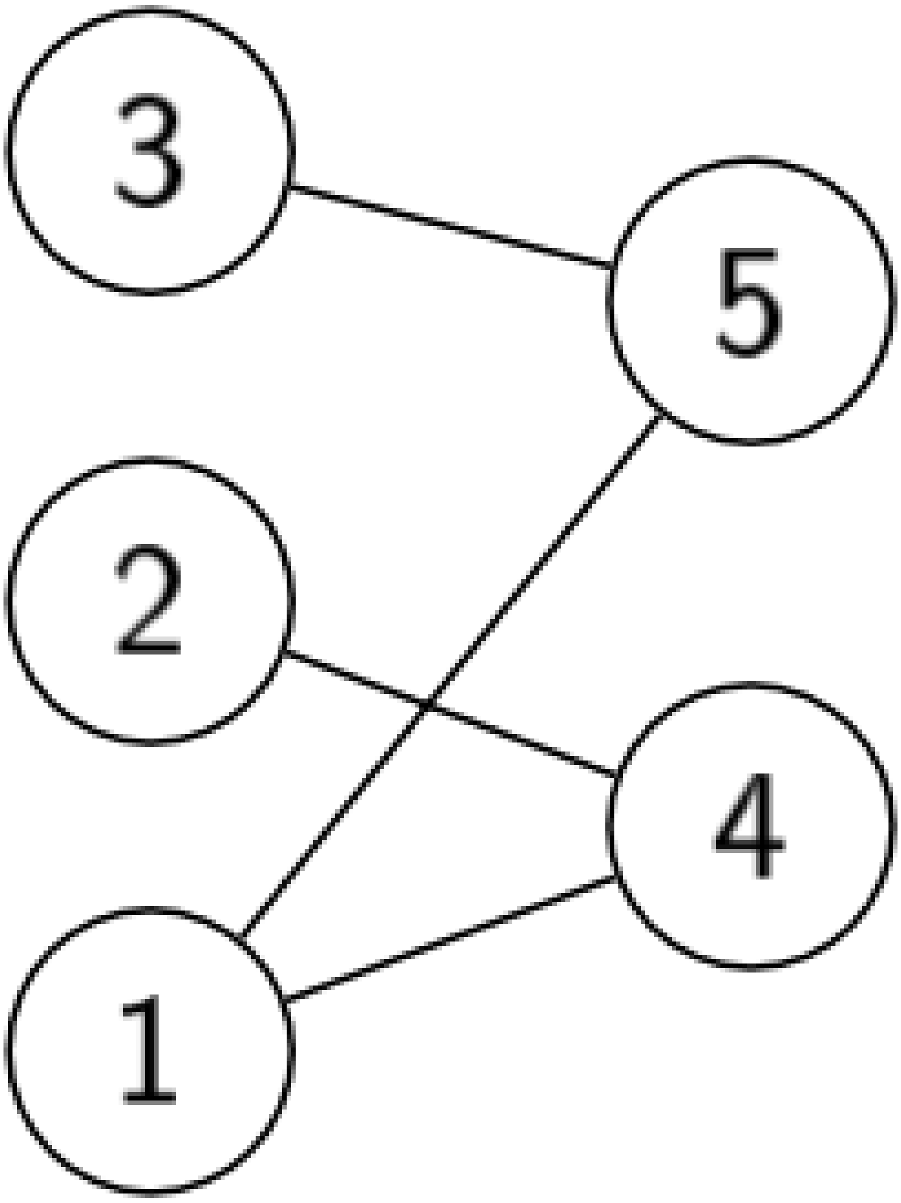


Nodes 1 and 2 are adjacent (unlike nodes 1 and 3)
Edge $\{1, 2\}$ is incident in nodes 1 and 2
Node 4 has degree 4

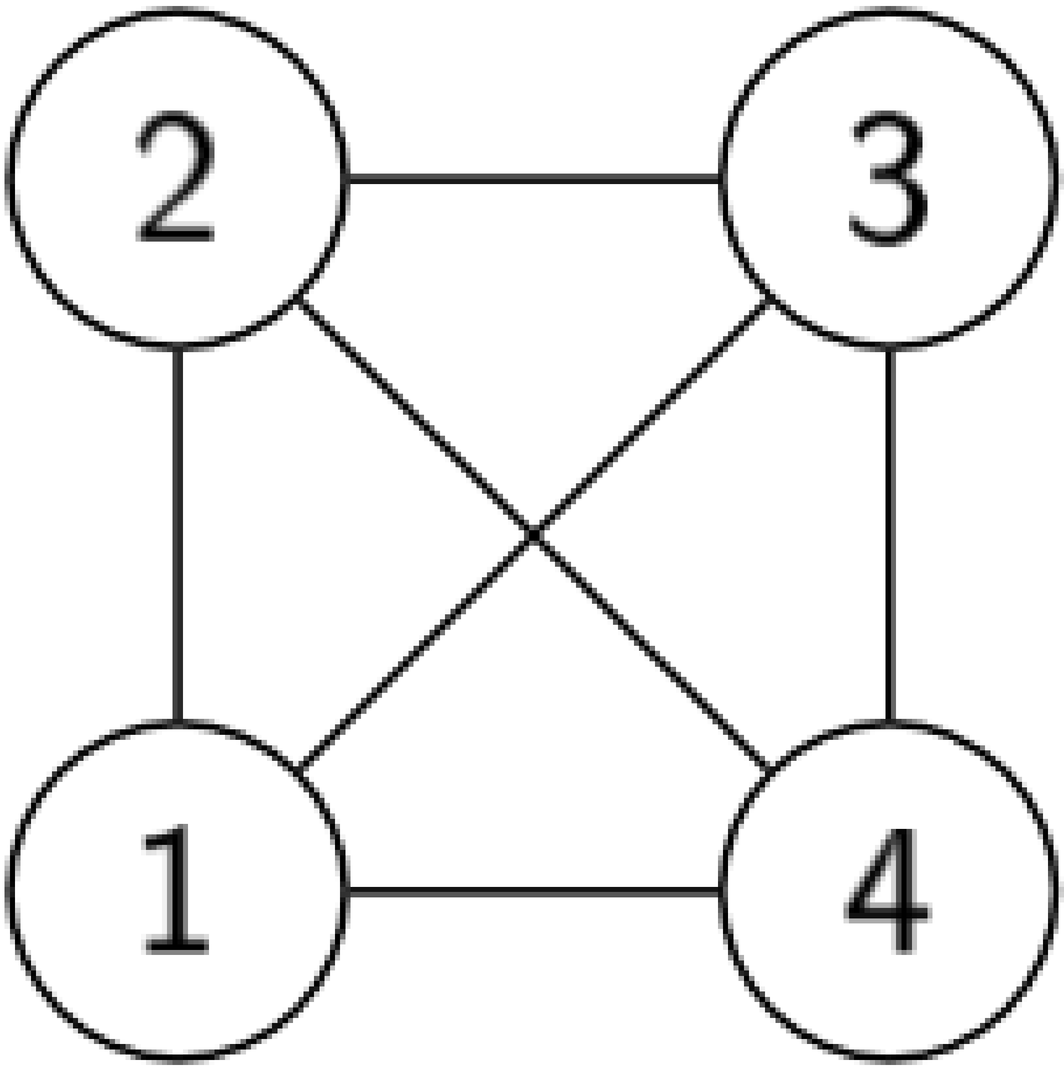


Node 1 has in-degree 0, out-degree 2

- An undirected graph $G = (N, E)$ with n nodes has $m \leq n(n-1)/2$ edges
- A directed graph $G = (N, A)$ with n nodes has $m \leq n(n-1)$ arcs
- A graph is **dense** if $m \approx n^2$, while it is **sparse** if $m \ll n^2$
- A graph is said to be **bipartite** if there's a partition $N = N_1 \cup N_2$, with $N_1 \cap N_2 = \emptyset$, such that no edge connects nodes in the same subset:



- A graph is **complete** if $E = \{\{v_i, v_j\} : v_i, v_j \in N \wedge i \leq j\}$:



Paths

Paths

A path from $v_1 \in N$ to $v_k \in N$ is a sequence of consecutive edges:

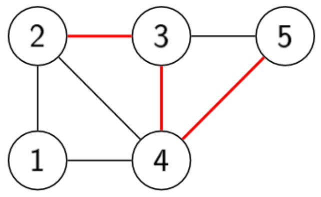
$$p = \langle \{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\} \rangle$$

with $(v_l, v_{l+1}) \in E$ for $l = 1, \ldots, k-1$.

Nodes u and v are **connected** if there is a path connecting them, while a graph is **connected** if u, v are connected for every $u, v \in N$.

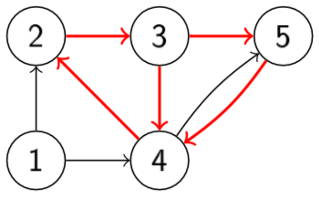
A directed graph is **strongly connected** if u, v are connected by a directed path for every $u, v \in N$.

Examples: Connected graph



$\langle \{2,3\}, \{3,4\}, \{4,5\} \rangle$ is a path from node 2 to node 5
Nodes 2 and 5 are connected

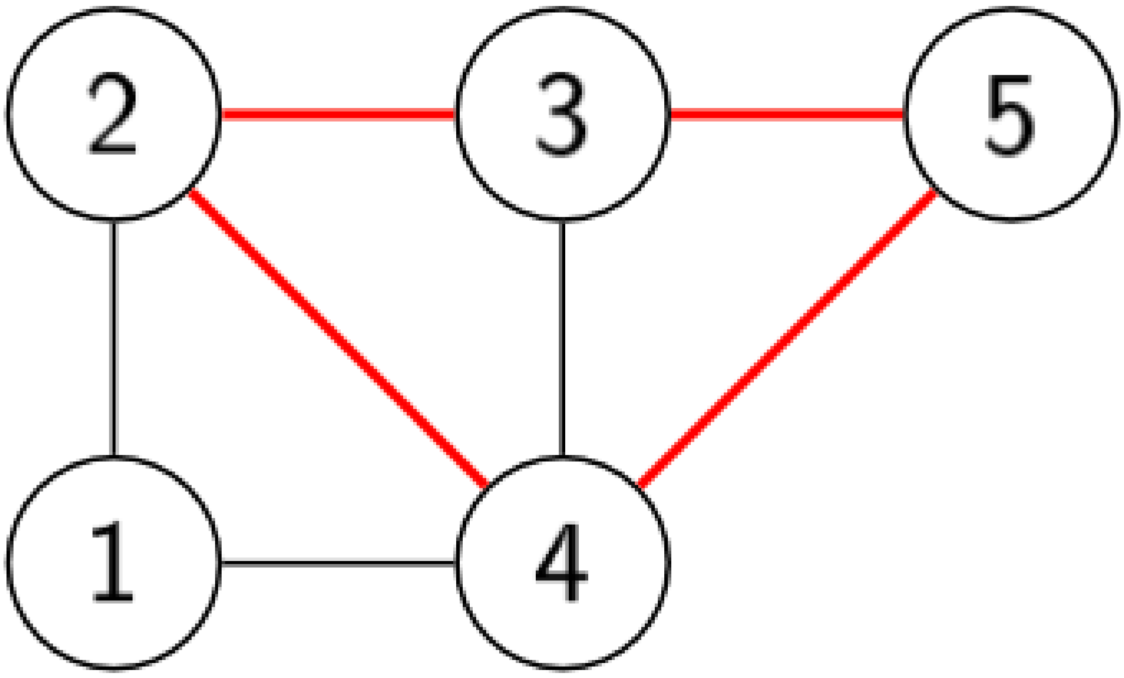
Not strongly connected graph



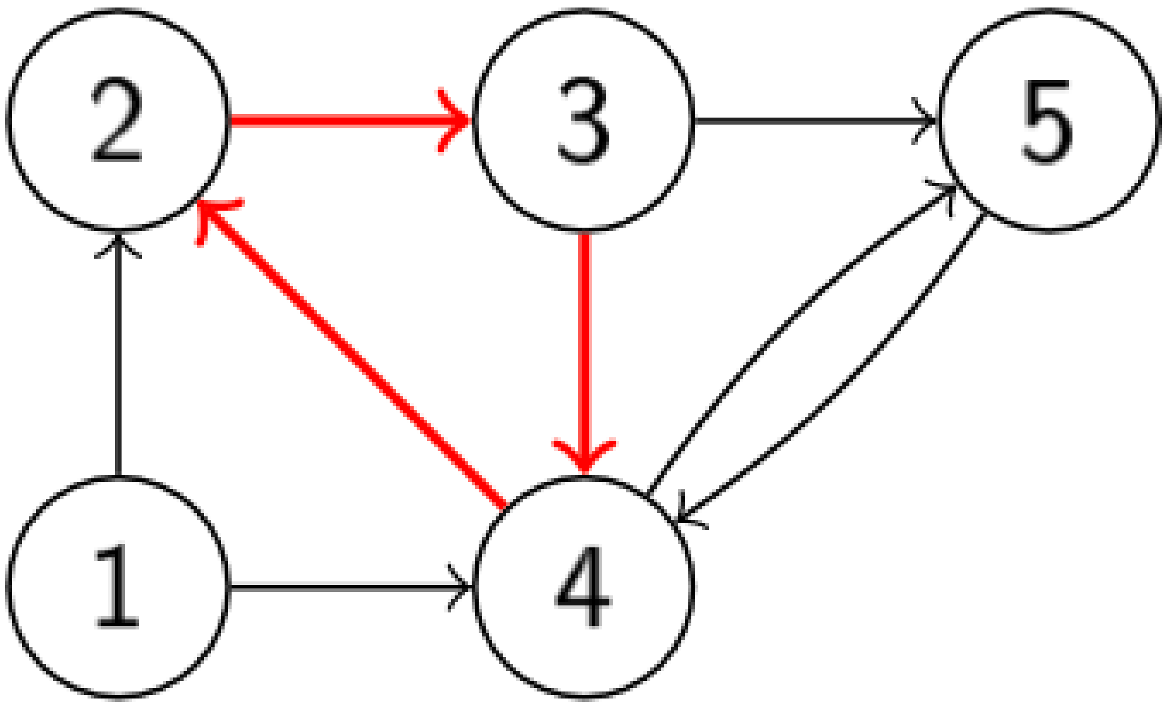
$\langle (3,5), (5,4), (4,2), (2,3), (3,4) \rangle$ is a directed path from node 3 to node 4

Cycles and circuits

In an undirected graph, a cycle is a path with $v_1 = v_k$:



In a directed graph, instead, a circuit is a path with $v_1 = v_k$:



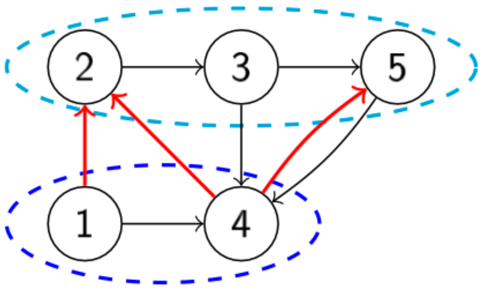
Cuts

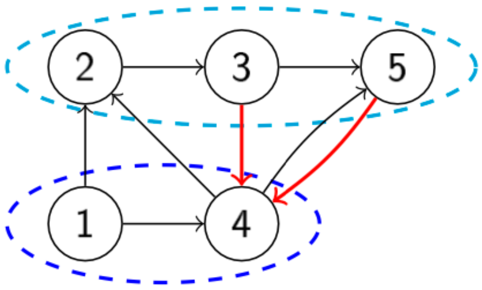
Given a directed graph $G = (N, A)$ and $S \subset N$, the **outgoing cut** induced by S is:

$$\delta^+(S) = \{(u, v) \in A : u \in S \wedge v \in N \setminus S\}$$

while the **incoming cut** induced by S is:

$$\delta^-(S) = \{(u, v) \in A : v \in S \wedge u \in N \setminus S\}$$





$\delta^+(\{1,4\}) = \{(1,2),(4,2),(4,5)\}$
$S = \{1,4\}$
$N \backslash S = \{2,3,5\}$

$\delta^-(\{1,4\}) = \{(3,4),(5,4)\}$
$S = \{1,4\}$
$N \backslash S = \{2,3,5\}$

For an undirected graph $G = (N, E)$ instead, where $S \subset N$, the **cut** induced by S is:

$$\delta(S) = \{(u, v) \in E : u \in S \wedge v \in N \setminus S\}$$

Graph representation

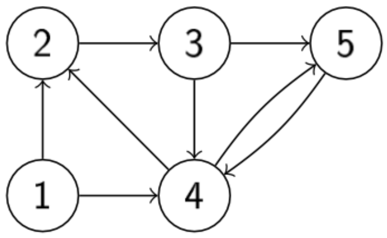
There are different ways to represent a graph.

Generally speaking, for dense, directed graphs, an **adjacency matrix** is used, where every element is defined like so:

$$a_{ij} = \begin{cases} 1, & (i,j) \in A \\ 0, & \text{otherwise} \end{cases}$$

For sparse directed graphs, a **list of successors** (or predecessors) can also be used.

Below is an example of both an adjacency matrix and a list of successors:



$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\begin{aligned} S(1) &= \{2,4\} \\ S(2) &= \{3\} \\ S(3) &= \{4,5\} \\ S(4) &= \{2,5\} \\ S(5) &= \{4\} \end{aligned}$$

The same representations can also be used for undirected graphs.

2.1.1. The graph reachability problem

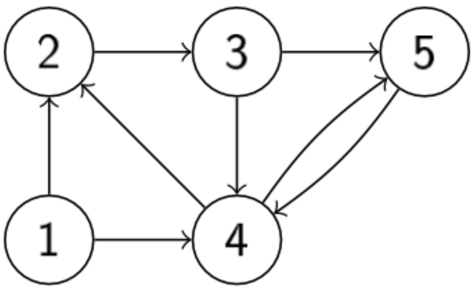
The reachability problem have the following structure:

> **Given a directed graph $G = (N, A)$ and a node s, determine all the nodes that are reachable from s.**

Our goal is to define an efficient algorithm that allows us to find all the nodes that are reachable from s.

Algorithm 1: FIFO queue

We could use a queue with First In-First Out policy Q containing the nodes reachable from s and not yet processed. For example:



$$\begin{aligned}
&Q = \{2\} && M = \emptyset \\
&Q = \{\cancel{2}\} && M = \{2\} \\
&Q = \{\cancel{3}\} && M = M \cup \{3\} \\
&Q = \{\cancel{4}, 5\} && M = M \cup \{4\} \\
&Q = \{\cancel{5}\} && M = M \cup \{5\} \\
&Q = \emptyset &&
\end{aligned}$$

$M = \{2, 3, 4, 5\}$ is the set of nodes that are reachable from $s = 2$.

We can write this algorithm with a sort of pseudocode like so:

```
$Q \leftarrow \{s\}; M \leftarrow \emptyset$
while $Q \neq \emptyset$ do
  $u \leftarrow$ node in $Q$; $Q \leftarrow Q \setminus \{u\}$
  $M \leftarrow M \cup \{u\}$
  for $v \in S(u)$ do
    if $v \notin M$ and $v \notin Q$ then $Q \leftarrow Q \cup \{v\}$
```

Note

If Q is managed as a FIFO queue, the nodes are explored by **breadth-first search**.

This algorithm stops when $\delta^+(M) = \emptyset$. When this is reached, $\delta^-(M)$ is the set of arcs with head node in M and tail node in $N \setminus M$.

Complexity

At each iteration of the `while` loop, the algorithm selects one node u from Q and inserts it in M. Then, for all nodes v directly reachable from u and not already in M or Q, v is inserted in Q.

Since each node u is inserted in Q at most once and each arc (u, v) is considered at most once, the overall complexity is:

$$O(n + m), \; n = |N|, m = |A|$$

Note

For dense graphs, it holds that:

$$m = O(n^2)$$

2.1.3. Subgraphs and trees

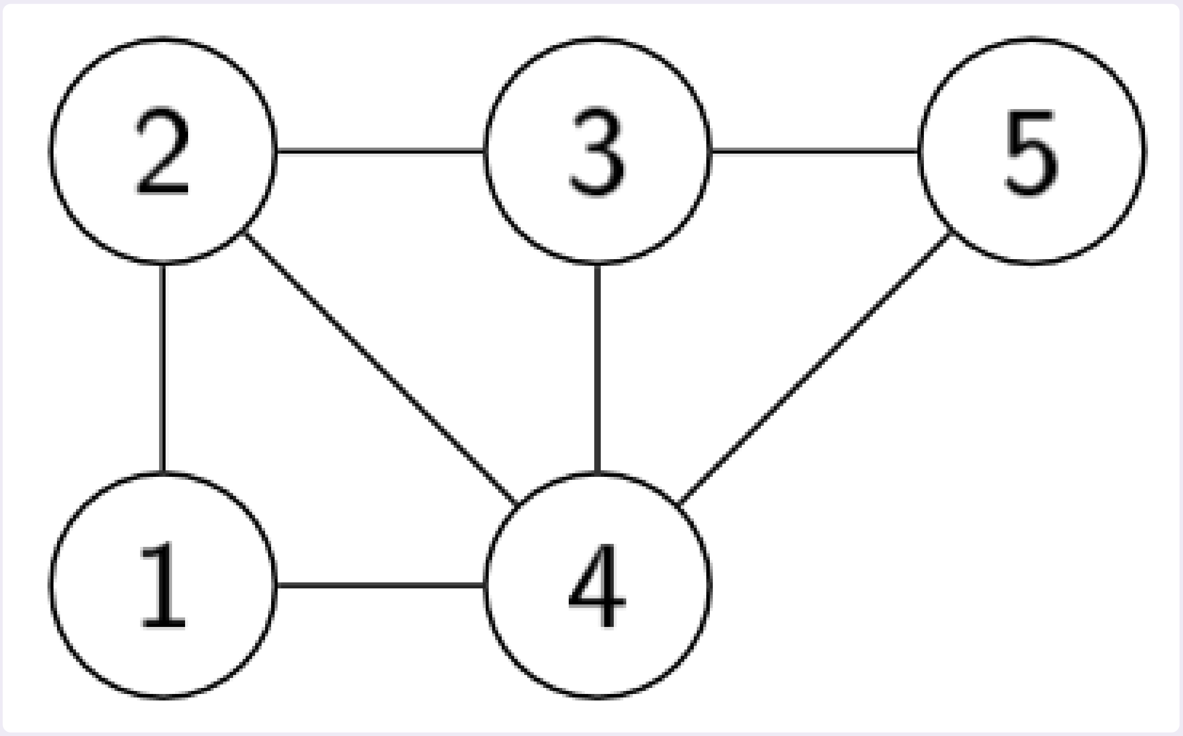
$G' = (N', E')$ is a **subgraph** of $G = (N, E)$ if $N' \subseteq N$ and $E' \subseteq E$.

A **tree** $G_T = (N', T)$ of G is a **connected, acyclic subgraph** of G. G_T is a **spanning tree** of G if it contains all the nodes in G ($N = N'$).

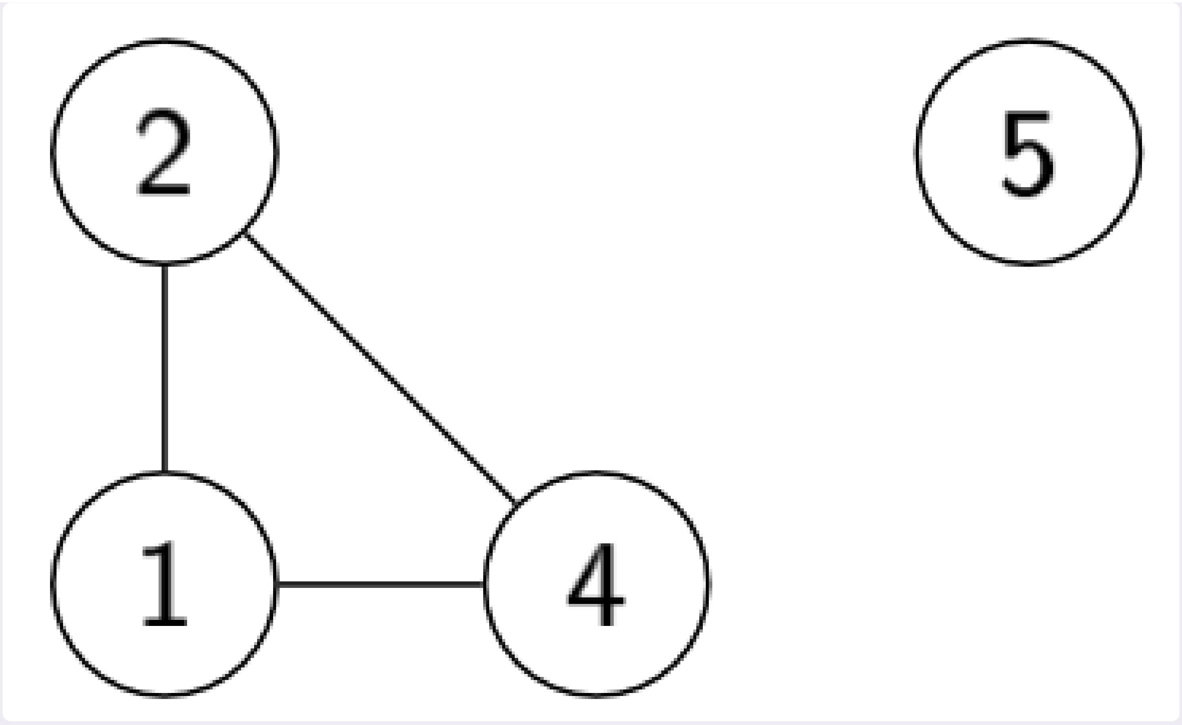
The leaves of a tree are all the nodes of degree one.

Example

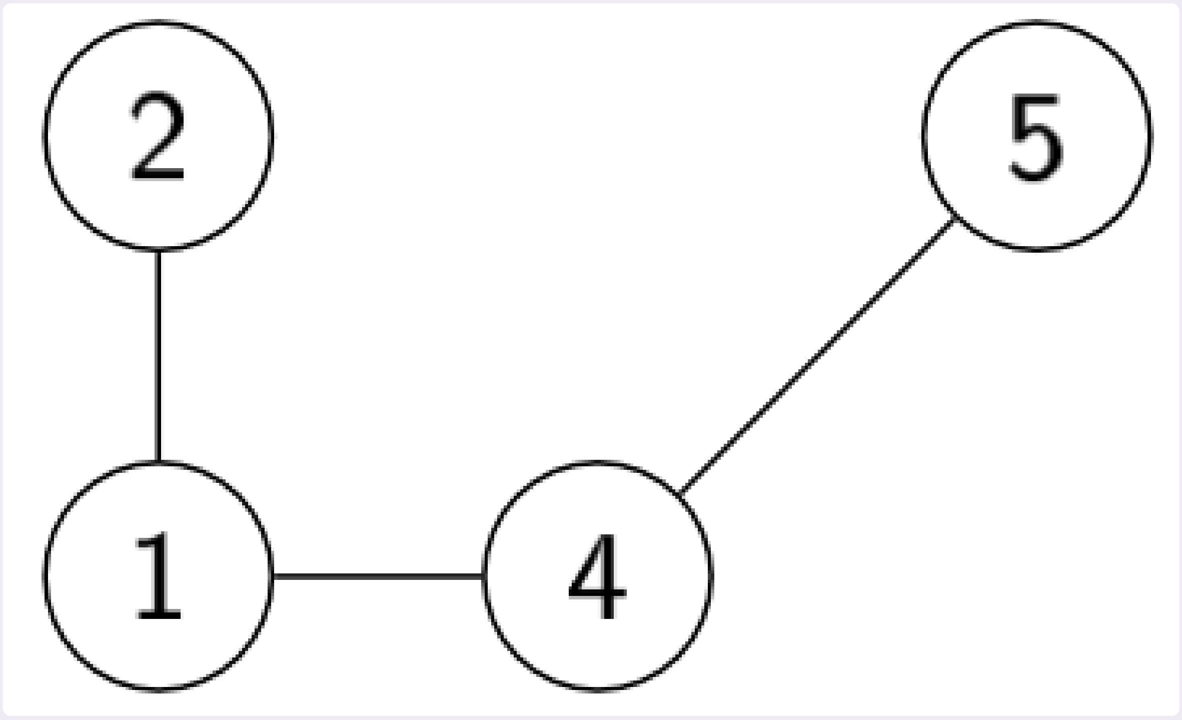
Given the following graph G:



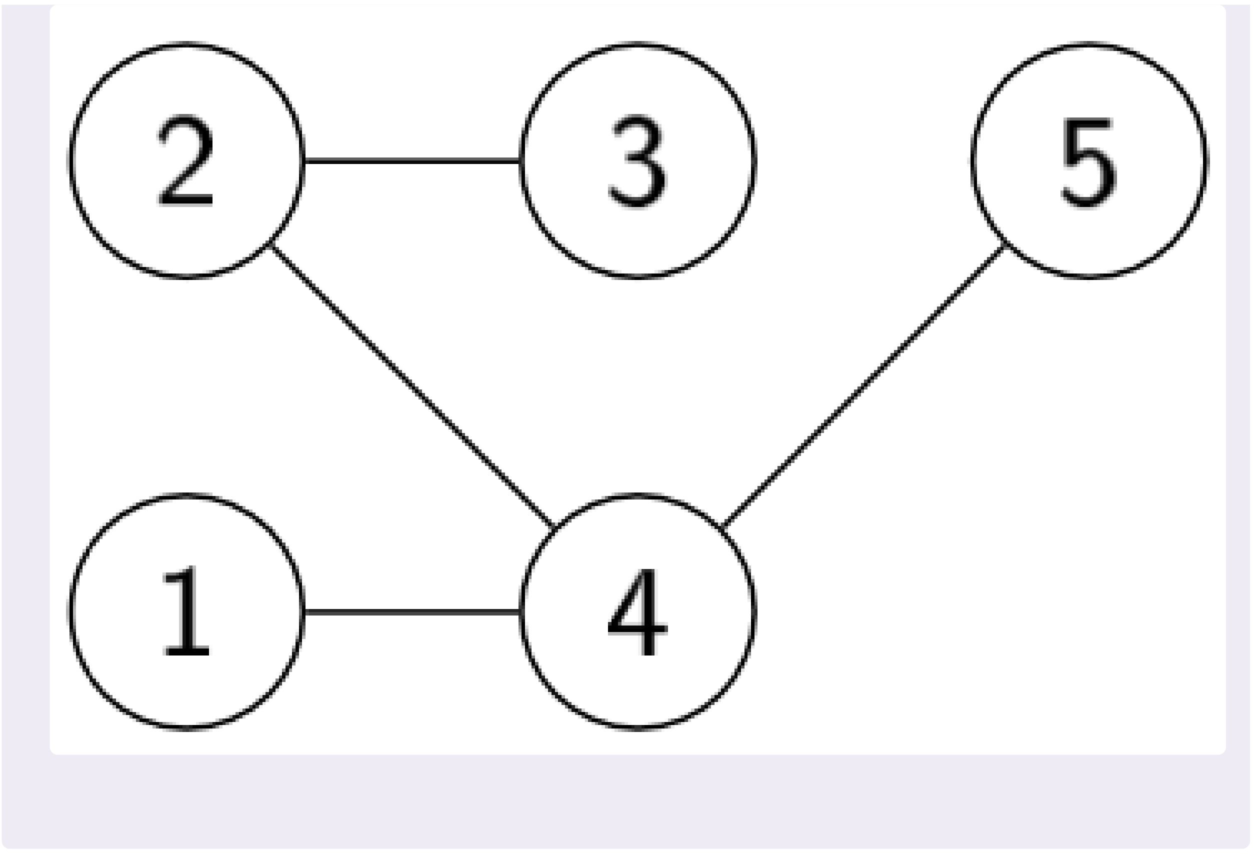
This is a **subgraph** of G:



This is a **tree** of *G*:



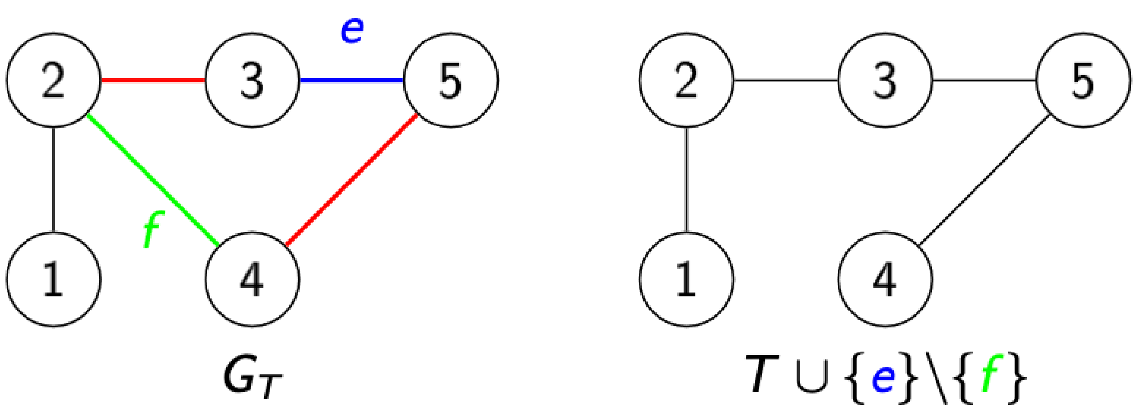
This is a **spanning tree** of *G*:



Properties of trees

Trees have some interesting properties:

1. Every tree with n nodes has $n - 1$ edges
2. Any pair of nodes in a tree is connected via a unique path
3. By adding a new edge to a tree, we create a unique cycle
4. **Exchange property**: let $G_T = (N, T)$ be a spanning tree of $G = (N, E)$. Consider an edge $e \notin T$ and the unique cycle C of $T \cup \{e\}$. For each edge $f \in C \setminus \{e\}$, the subgraph $T \cup \{e\} \setminus \{f\}$ is also a spanning tree of G:



2.2. Optimal cost spanning trees

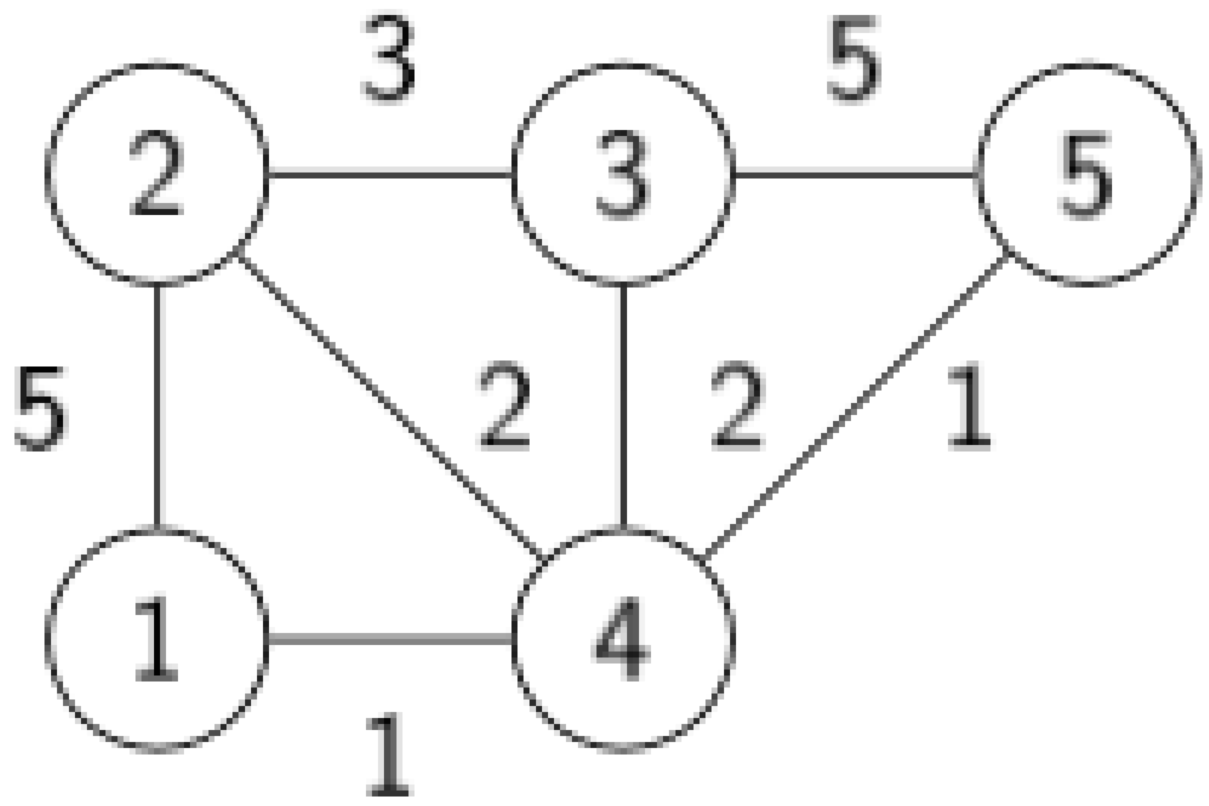
Optimal cost spanning trees have a number of applications, like:

- Network design
- IP network protocols
- Compact memory storage (DNA)

Let's take this problem as an example:

Design a communication network so as to connect n cities at a minimum total cost

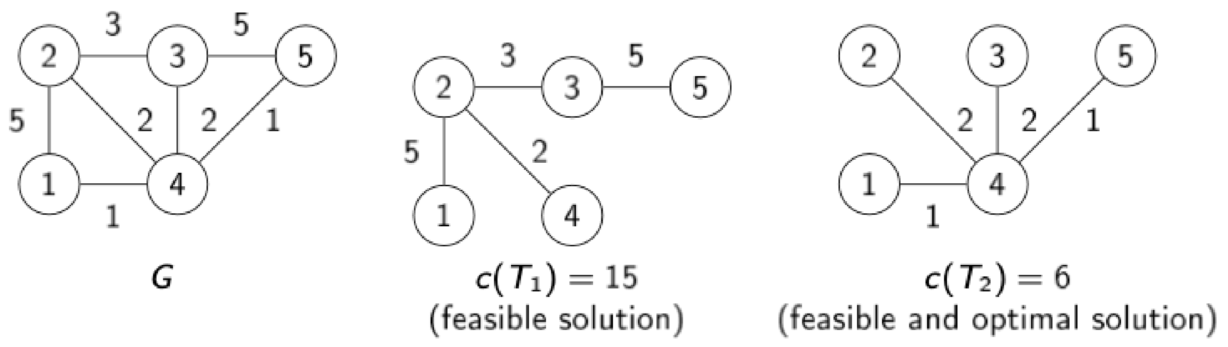
We can model the network through a graph like the following:



In order to design a communication network like the one described, we require the two properties below:

- Each pair of cities must communicate, hence we need a connected subgraph containing all the nodes of the original graph
- The network must be cost effective, so we require that the subgraph has no cycles

One solution to this problem is reported in the following image:



We can now enunciate **Cayley's theorem**:

Cayley's theorem

A complete graph with n nodes, with $n \geq 1$, has n^{n-2} spanning trees

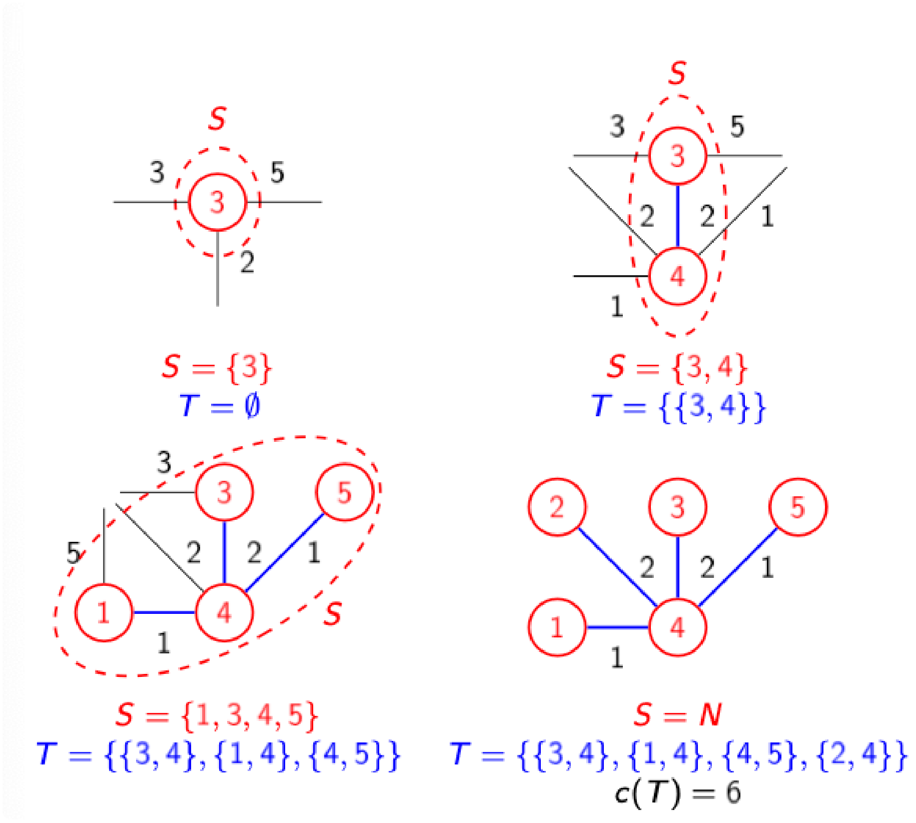
2.2.1. Prim's algorithm

One algorithm to build a spanning tree starting from a graph is Prim's algorithm, which creates a spanning tree iteratively. It works as follows:

Prim's algorithm

Start from initial tree (S, T) with $S = \{u\}$ and $T = \emptyset$ (u can be any node in N). At each step, add to the current partial tree (S, T) an edge of minimum cost among those which connect a node in S to a node in $N \setminus S$.

Recalling the example from before, here is the application of Prim's algorithm to it:



Prim's algorithm is **exact**[1]. This does not depend on the choice of the first node nor on the selected edge of minimum cost in $\delta(S)$.

We can also show that each selected edge belongs to a minimum cost spanning tree through the **cut property**:

Cut property

Let F be a partial tree contained in an optimal tree of G. Consider $e = \{u, v\} \in \delta(S)$ of minimum cost, then there exists a minimum cost spanning tree of G containing e.

Furthermore, we can state that Prim's algorithm is **greedy**[2]: at each step, a minimum cost edge is selected among those in the cut $\delta(S)$ induced by the current set of nodes S.

Note

For most discrete optimisation problems, greedy-type algorithms yield a feasible solution with **no guarantee of optimality**

Let's look at an implementation of Prim's algorithm with complexity $O(n^2)$. Let's create the data structure needed like so:

- k: number of edges selected so far
- Subset $S \subseteq N$ of nodes incident to the selected edges
- Subset $T \subseteq E$ of selected edges

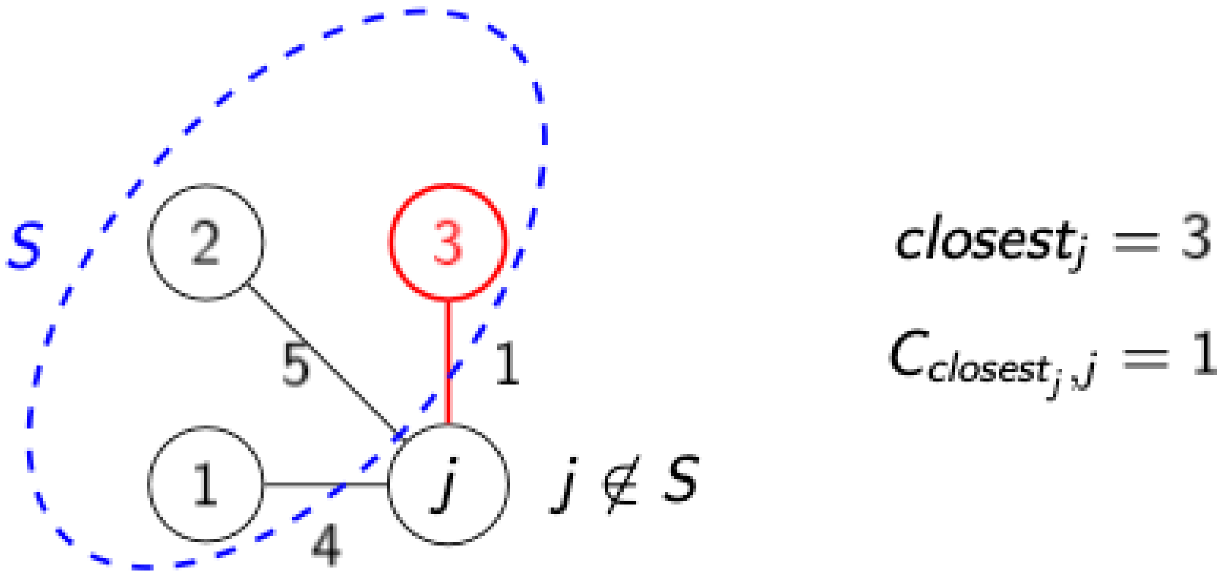
Let's also define C_j like so:

$$C_j = \begin{cases} \min\{c_{ij} : i \in S\}, & j \notin S \\ +\infty, & \text{otherwise} \end{cases}$$

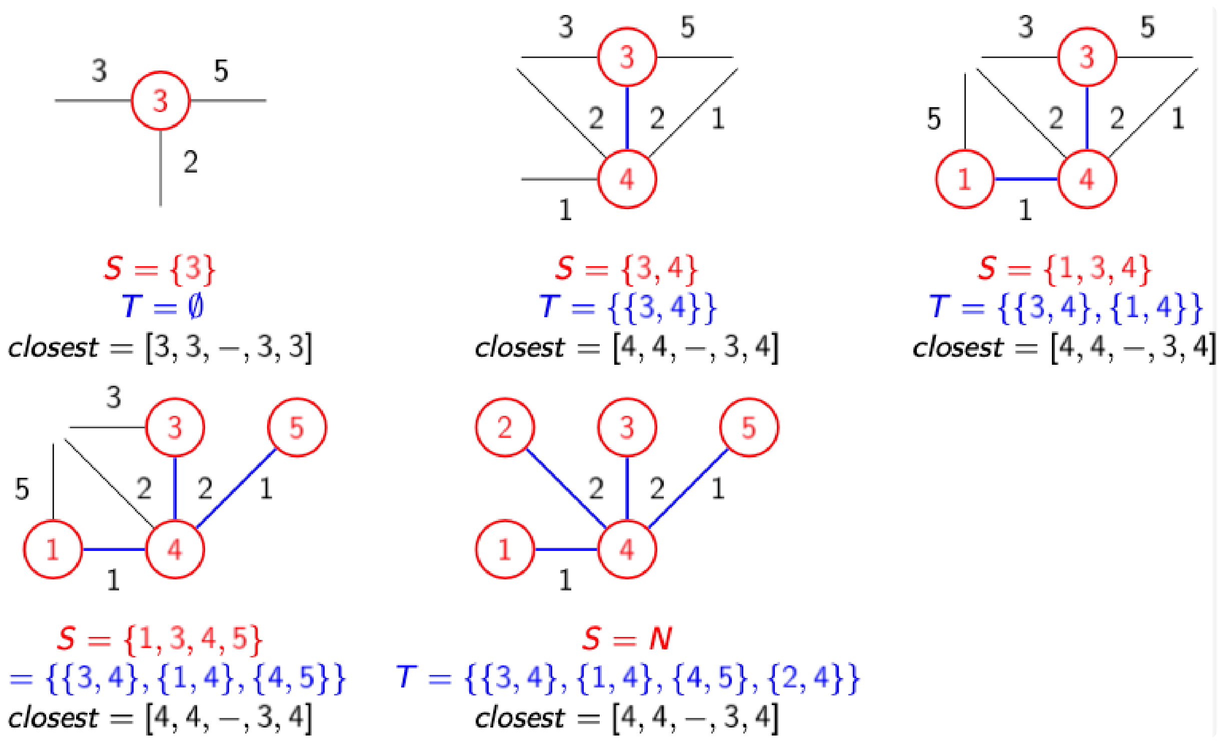
and closest_j like so:

$$\text{closest}_j = \begin{cases} \arg\min\{c_{ij} : i \in S\}, & j \notin S \\ \text{"predecessor" of } j \text{ in the minimum spanning tree}, & j \in S \end{cases}$$

For example, if node 3 is selected from the graph of the example, we have:

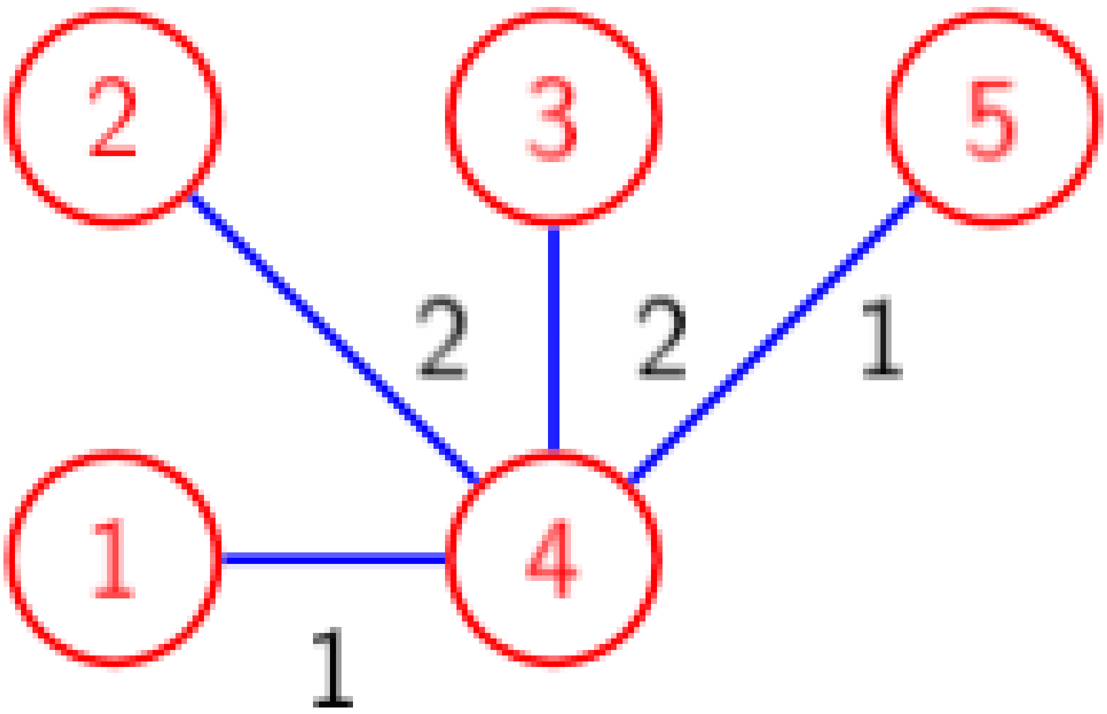


Here is how the algorithm would execute starting from node 3:


$S = \{3\}$
$T = \emptyset$
$closest = [3, 3, -, 3, 3]$
$S = \{3, 4\}$
$T = \{\{3, 4\}\}$
$closest = [4, 4, -, 3, 4]$
$S = \{1, 3, 4\}$
$T = \{\{3, 4\}, \{1, 4\}\}$
$closest = [4, 4, -, 3, 4]$
$S = \{1, 3, 4, 5\}$
$T = \{\{3, 4\}, \{1, 4\}, \{4, 5\}\}$
$closest = [4, 4, -, 3, 4]$
$S = N$
$T = \{\{3, 4\}, \{1, 4\}, \{4, 5\}, \{2, 4\}\}$
$closest = [4, 4, -, 3, 4]$

The minimum spanning tree found by Prim's algorithm consists of the $n-1$ edges $\{\text{closest}_j, j\}$, with $j = 1, 2, \ldots, n$ and $j \neq u$.

In the example, since $\text{closest} = [4, 4, -, 3, 4]$, a minimum cost spanning tree consists of the edges $\{4, 1\}, \{4, 2\}, \{3, 4\}, \{4, 5\}$:



Here's the full algorithm in its $O(n^2)$ version:

```
$T \leftarrow \emptyset$; $S \leftarrow \{u\}$
for $j \in N \setminus S$ do
    $C_j \leftarrow c_{uj}$
    $\text{closest}_j \leftarrow u$

for $k = 1, \dots, n-1$ do
    $\min \leftarrow +\infty$
    for $j = 1, \dots, n$ do
        if $j \notin S$ and $C_j < \min$ then
            $\min \leftarrow C_j$; $v \leftarrow j$

    $S \leftarrow S \cup \{v\}$; $T \leftarrow T \cup \{\{\text{closest}_v, v\}\}$
    for $j = 1, \dots, n$ do
        if $j \notin S$ and $c_{vj} < C_j$ then
            $C_j \leftarrow c_{vj}$ : $\text{closest}_j \leftarrow v$
```

For sparse graphs, where $m \ll n(n-1)/2$, a more sophisticated data structure leads to a $O(m \log n)$ complexity.

2.2.2. Optimality condition

Let's define a property of edges:

Cost decreasing edges

Given a spanning tree T, an edge $e \notin T$ is **cost decreasing** if, when e is added to T, it creates a cycle C with $C \subseteq T \cup \{e\}$ and $\exists f \in C \setminus \{e\}$ such that $c_e < c_f$

Now we can enunciate the **tree optimality condition** theorem:

Tree optimality condition

A tree T is of minimum total cost if and only if **no cost-decreasing edge exists**

To verify the optimality condition, we can use the **optimality test**, which checks that each $e \in E \setminus T$ is not a cost-decreasing edge.

2.3. Optimal paths

2.3.1. Shortest path problem

Given a directed graph $G = (N, A)$ with a cost $c_{ij} \in \mathbb{R}$ for each arc $(i, j) \in A$ and two nodes s and t, determine a **minimum cost path** from s to t.

c_{ij} represents the cost associated with the arc $(i, j) \in A$, node s is called **origin**, and node t is called **destination**.

2.3.2. Dijkstra's algorithm

In order to enunciate Dijkstra's algorithm, we must first define what a **simple path** is:

Simple path

A path $\langle (i_1, i_2), (i_2, i_3), \dots, (i_{k-1}, i_k) \rangle$ is **simple** if **no node is visited more than once**.

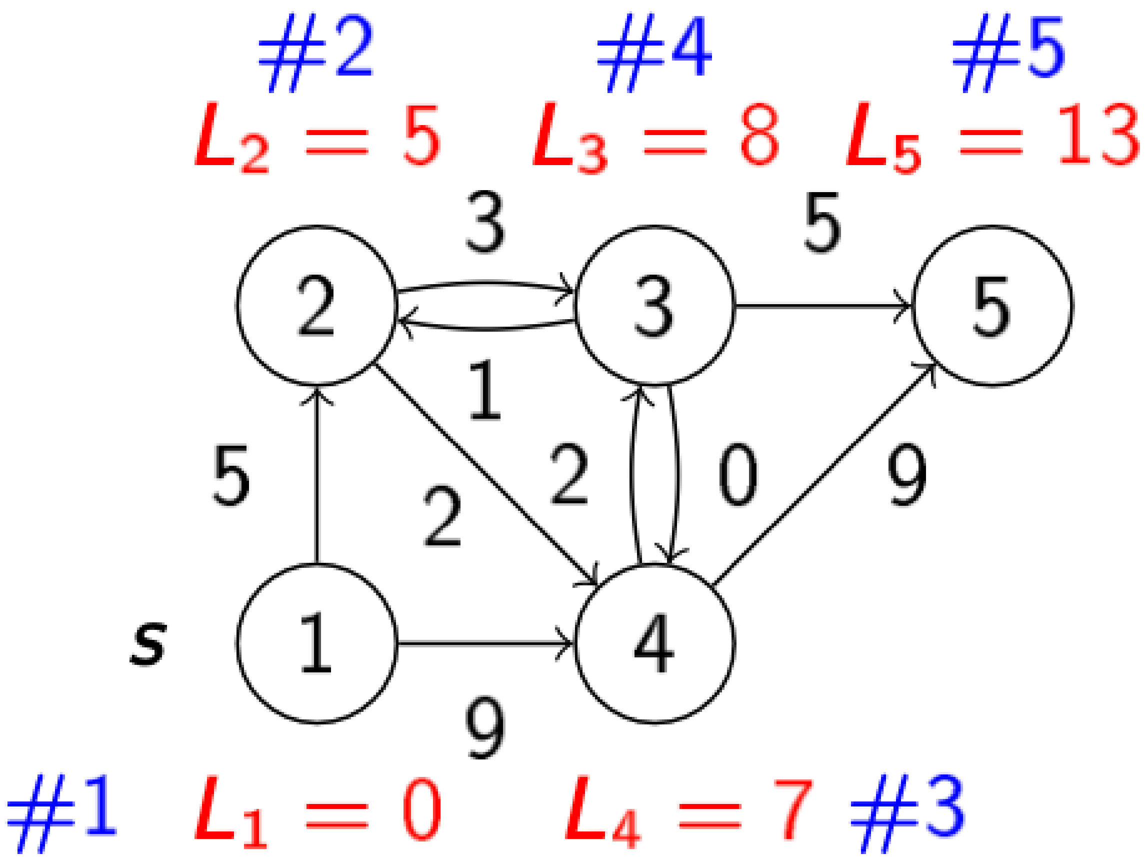
We can also state a useful property:

Property 1

If $c_{ij} \geq 0$ for all $(i, j) \in A$, there is at least one shortest path which is simple.

If we assume that a graph has non-negative costs for each of its arcs, then we can use Dijkstra's algorithm to find the shortest paths from a given node s to all other nodes of the same graph.

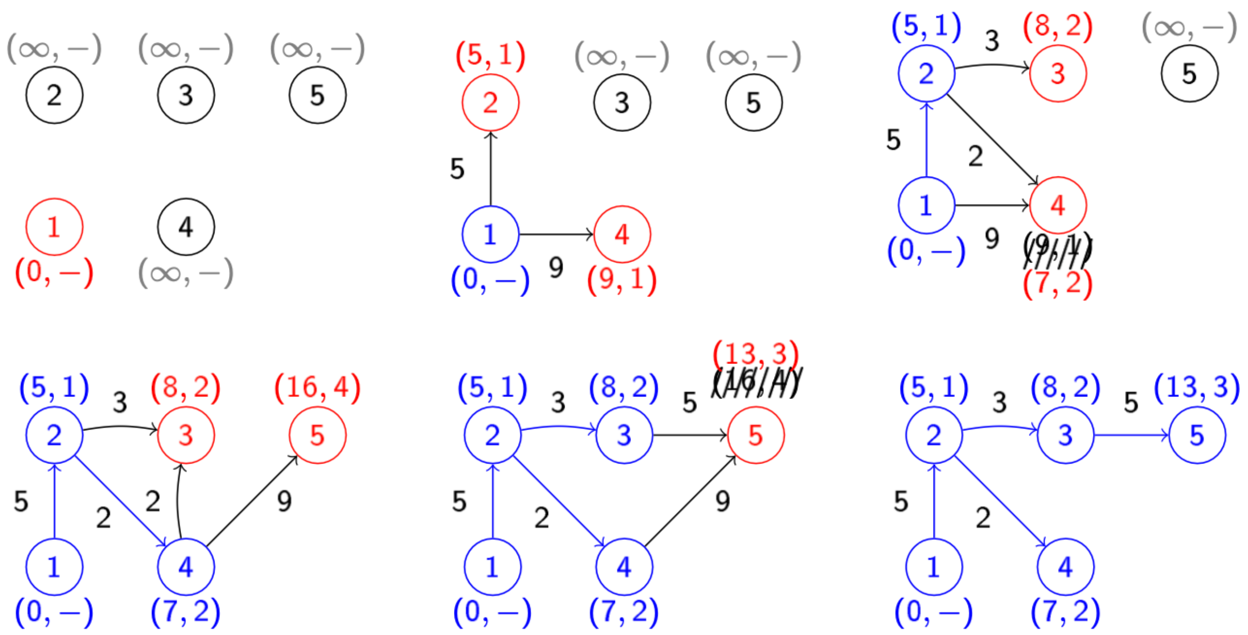
Let's see an example of application of Dijkstra's algorithm with the following graph:



Now, we associate two labels to each node which have the form (L_j, pred_j), where:

- L_j is the cost of the shortest path from s to j
- pred_j is the predecessor of j in a shortest path from s to j

The algorithm proceeds like so:



The data structure associated to Dijkstra's algorithm is as follows:

- $S \subseteq N$: subset of nodes whose labels are **permanent** (cannot be changed anymore by the algorithm)
- $\overline{S} = (N \setminus S) \subseteq N$: subset of nodes with **temporary** labels (they can still be changed by some iteration of the algorithm)
- $L_j = \begin{cases} \text{cost of a shortest path from } s \text{ to } j, & j \in S \\ \min\{L_i + c_{ij} : (i,j) \in \delta^+(S)\}, & j \notin S \end{cases}$
- $\text{pred}_j = \begin{cases} \text{predecessor of } j \text{ in the shortest path from } s \text{ to } j, & j \in S \\ u \text{ such that } L_u + c_{uj} = \min\{L_i + c_{ij} : i \in S\}, & j \notin S \end{cases}$

The complexity of Dijkstra's algorithm depends on how the arc (u, v) is selected among those of the current cut $\delta^+(S)$:

- If all m arcs are scanned, the overall complexity would be $O(nm)$
- If all labels L_j are determined by appropriate updates, the overall complexity is $O(n^2)$

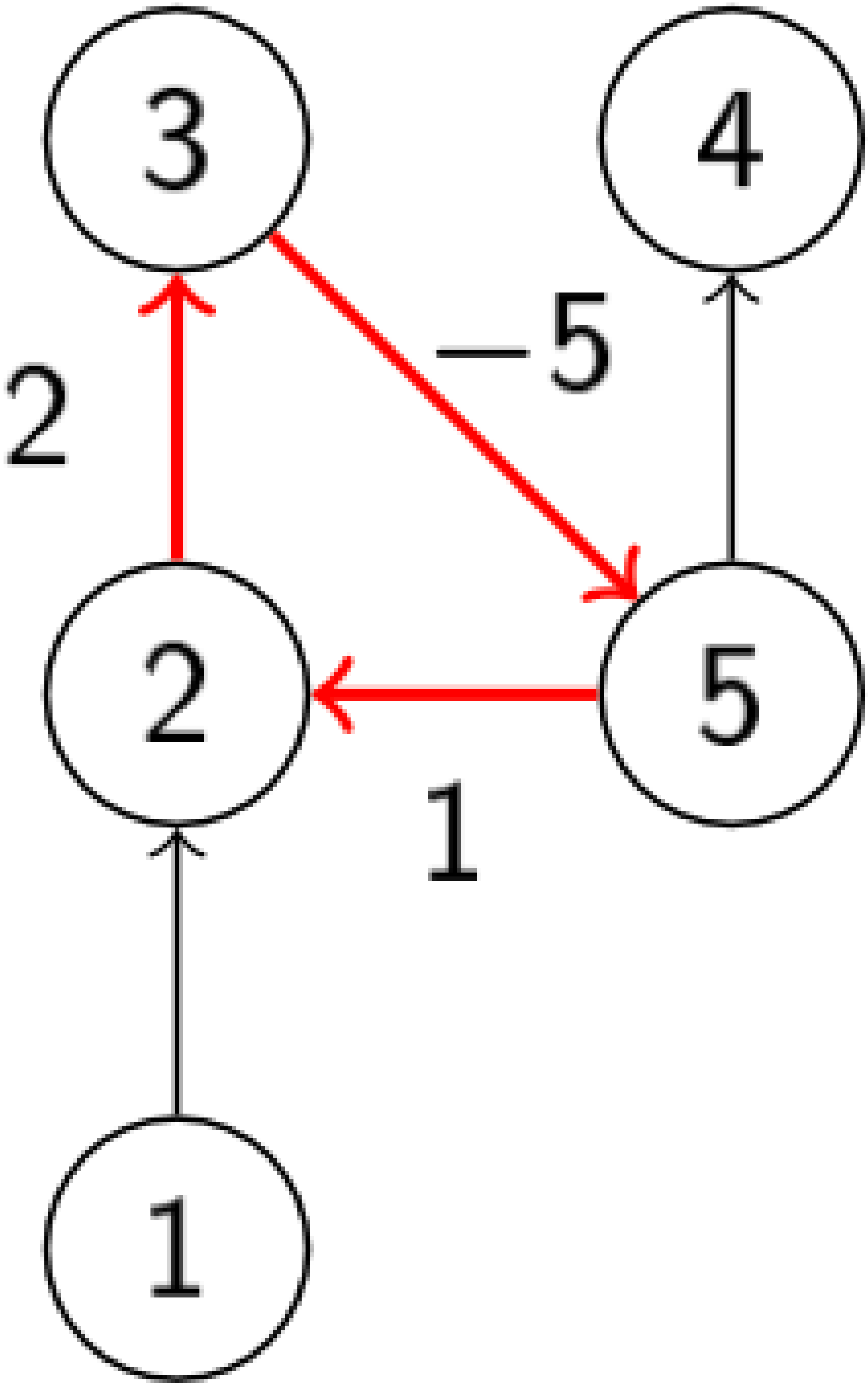
We can say that Dijkstra's algorithm is **exact**.

Remarks

- A set of shortest paths from s to all the nodes j can be retrieved via the vector of predecessors
- The union of a set of shortest paths from s to all the other nodes is an arborescence rooted at s, called **shortest path trees** (which have nothing to do with minimum cost spanning trees)
- Dijkstra's algorithm does not work with negative cost arcs

2.3.3. Shortest path problem with negative costs: Floyd-Warshall's algorithm

If a graph G contains a circuit of negative cost, the shortest path problem may not be well-defined. The following graph contains a circuit of cost -2, for example:


3
4
2
−5
2
5
1
1

In order to find out whether a graph contains circuits of negative cost, Floyd-Warshall's algorithm is employed to detect them. Furthermore, the algorithm also provides a set of shortest paths between **all pairs of nodes**, even when there are arcs with negative cost.

Floyd-Warshall's algorithm uses two $n \times n$ matrices as a data structure, respectively D and P, whose elements correspond to:

- d_{ij}: cost of a shortest path from i to j
- p_{ij}: predecessor of j in that shortest path from i to j

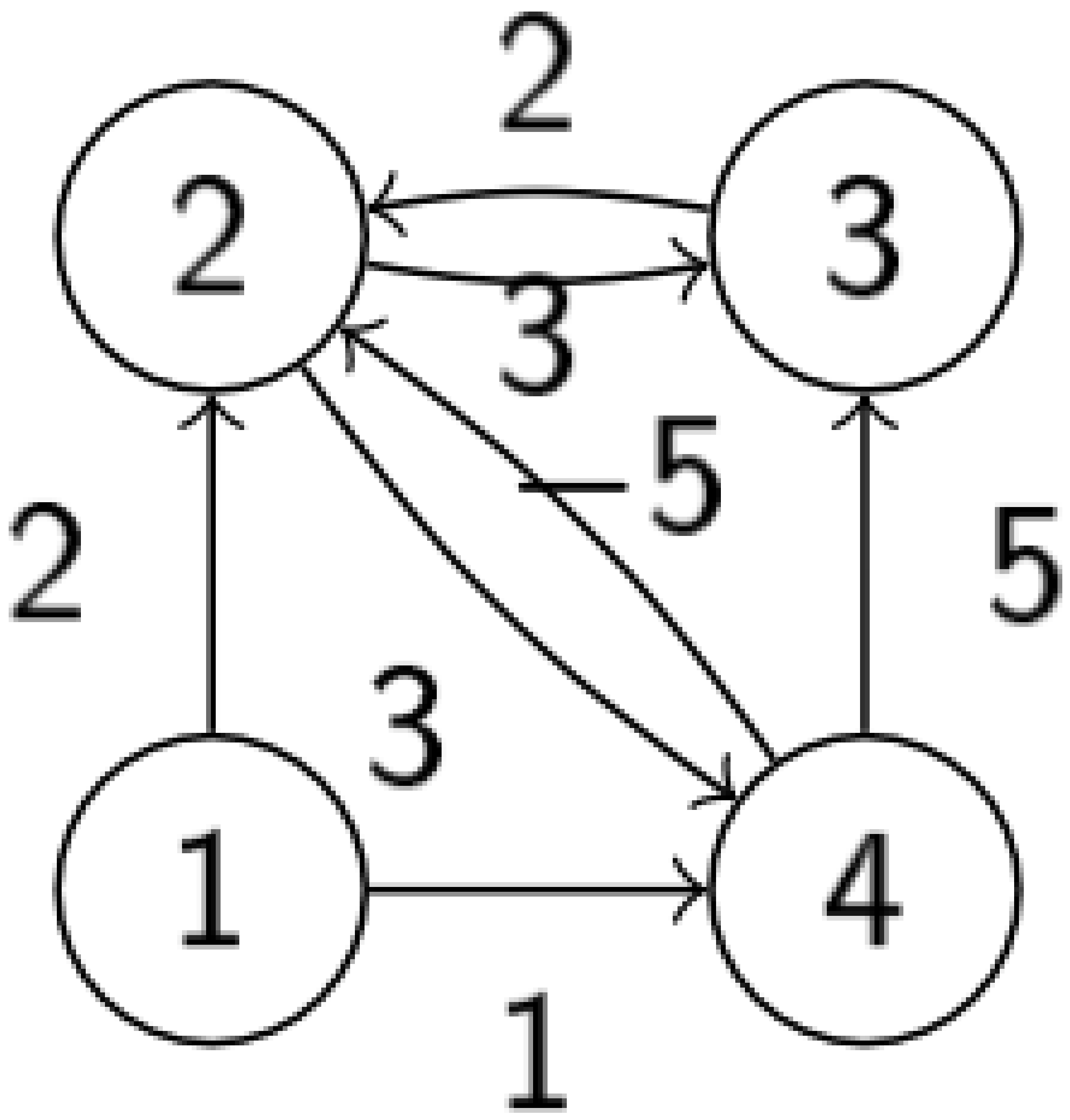
The elements in the D matrix are initialised according to this formula:

$$d_{ij} = \begin{cases} 0, & i = j \\ c_{ij}, & i \neq j \wedge (i, j) \in A \\ +\infty, & \text{otherwise} \end{cases}$$

The elements in the P matrix are initialised like so:

$$p_{ij} = i,\ i \in N$$

For example, starting from this graph:



We can retrieve the following matrices:

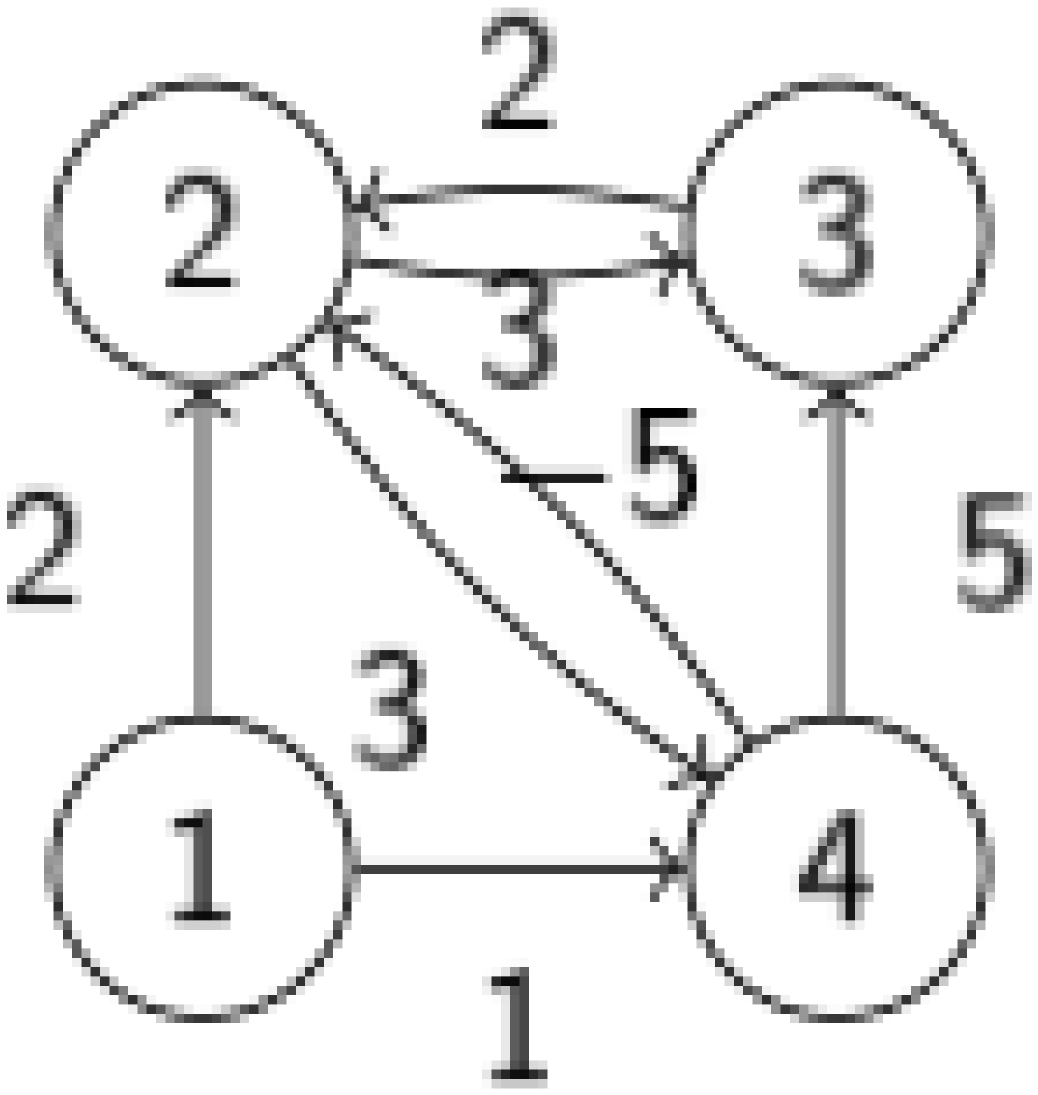
$$D = \begin{bmatrix} 0 & 2 & \infty & 1 \\ \infty & 0 & 3 & 3 \\ \infty & 2 & 0 & \infty \\ \infty & -5 & 5 & 0 \end{bmatrix}, \ P = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{bmatrix}$$

A fundamental operation in the Floyd-Warshall algorithm is the **triangular operation**, which works like this:

Triangular operation

For each pair of nodes i, j with $i \neq u$ and $j \neq u$ (including the case $i = j$), check whether when going from i to j it is more convenient to go via u.
In short: if $d_{iu} + d_{uj} < d_{ij}$, then $d_{ij} \leftarrow d_{iu} + d_{uj}$

The algorithm always halts once a negative cost circuit is found. In the following graph:



We find a negative cost circuit at iteration $u = 2$ of the algorithm: since $d_{44} = 0$ but $d_{42} + d_{24} = -2$ and still manages to start from node 4 and finish there too, a negative cost circuit is found.

We can say that Floyd-Warshall's algorithm is exact, just like Dijkstra's.

About complexity: since in the worst case the triangular operation is executed for all nodes u and for each pair of nodes i and j, the overall complexity is $O(n^3)$.

2.3.4. Optimal paths in directed acyclic graphs

To freshen our memory, let's enunciate the definition of *directed acyclic graph*:

> **Directed acyclic graph**
>
> A directed graph $G = (N, A)$ is **acyclic** (also known as DAG) if it contains **no circuits**.

A common problem entailing DAGs is the **shortest path** problem: given a DAG with a cost for each arc and nodes s and t, determine a **shortest** path from s to t. Its dual problem, the **longest path** problem, is also very common.

In order to find the longest or shortest path in DAGs, we can use one of their properties: the nodes of any DAG can be ordered **topologically**, that is, indexed so that for each arc $(i, j) \in A$ we have $i < j$. The topological order property can be exploited in a very efficient **dynamic programming** algorithm to find shortest or longest paths.

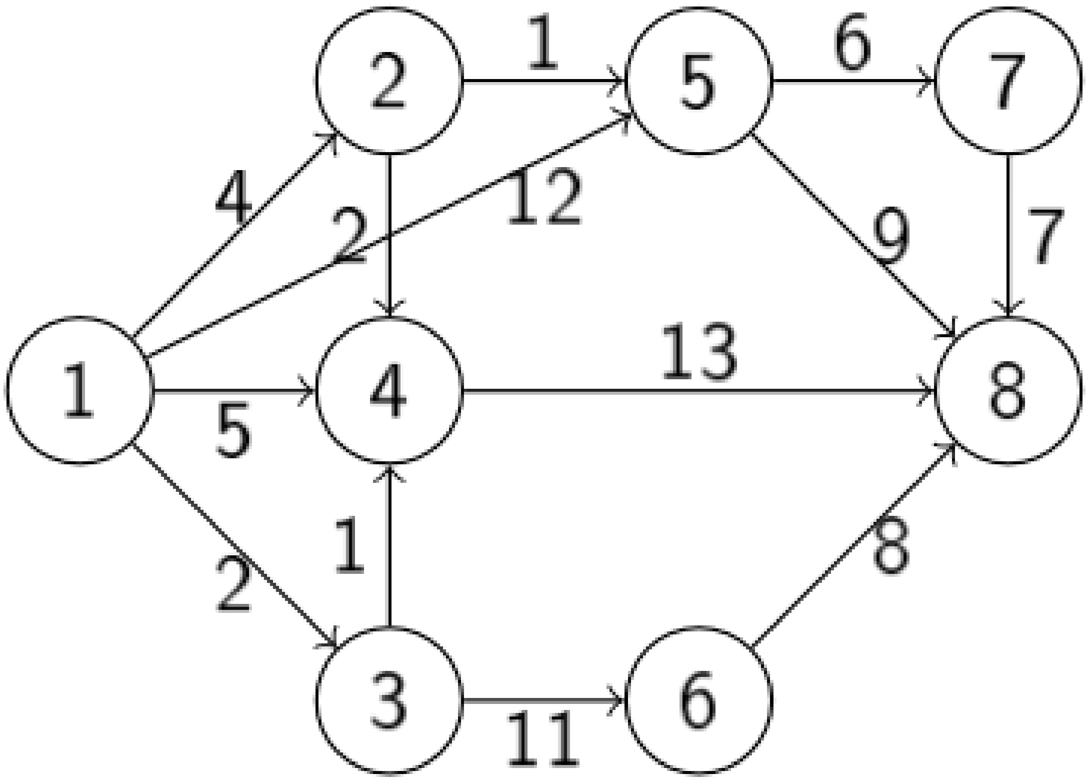
First, let's take a look at how to order a DAG topologically. If we represent a graph through its list of predecessors and successors for each node, then:

1. We assign the smallest positive integer not yet assigned to a node $v \in N$ with $\delta^-(v) = \emptyset$
2. We delete the node v with all its incident arcs
3. We repeat from step 1 until there are no more nodes in the current subgraph

Using this algorithm to order a graph topologically is an action of complexity $O(n + m)$, where n is the number of nodes and m is the number of arcs.

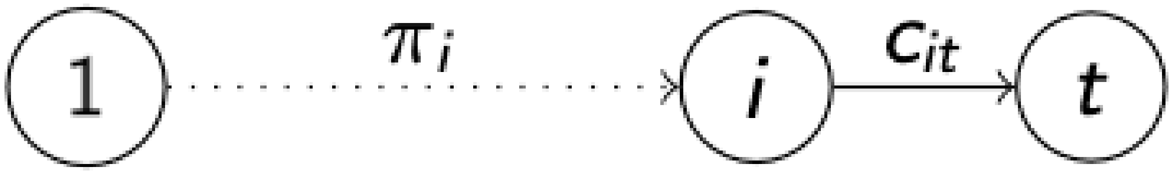
Dynamic Programming for shortest paths in DAGs

Let's try to solve a problem. Given the following DAG:



Given its topological ordering of nodes, how can we find a shortest path from node 1 to node 8?

In order to solve this problem, we can use the **Dynamic Programming algorithm**: any shortest path π_t from node 1 to t with at least two arcs can be subdivided in two parts: π_i and (i, t), where π_i is a **shortest subpath** from s to i.



In practice

Every path to a given node can be subdivided in a subpath from the source node to the penultimate node and the path from that node to the destination node.

For each node $i = 1, \ldots, t$, let L_i be the cost of a shortest path from 1 to i. Then:

$$L_t = \min_{(i,t)\in\delta^-(t)} \left(L_i + c_{it}\right)$$

Where the minimum is taken over all possible predecessors i of t.

If the graph G is a DAG and its nodes are topologically ordered, the **only possible predecessors** of t in a shortest path π_t from 1 to t are those with index $i < t$. Thus:

$$L_t = \min_{i<t} \left(L_i + c_{it}\right)$$

This is not true in a graph with circuits, since any node that is not t can be a predecessor of t in that case.

The complexity of this algorithm can be calculated in two parts:

- The topological ordering of the nodes is an operation of complexity $O(n+m)$
- The computation of every shortest subpath considers every node and arc exactly once, so its complexity is $O(n+m)$

Thus, the overall complexity of the Dynamic Programming algorithm is $O(n+m)$.

This algorithm can be easily modified for **longest paths** in DAGs like so:

$$L_t = \max_{i<t} \left(L_i + c_{it}\right)$$

We can say that the Dynamic Programming algorithm, both in the shortest path and longest path flavours, is **exact**. This is due to the **optimality principle**:

Optimality principle

For any shortest (respectively, longest) path π_j from 1 to j, there exists $i < j$ such that π_j can be subdivided into two parts: π_i and (i, j), where subpath π_i is of minimum (respectively, maximum) length from 1 to i.

2.3.5. Project planning

To help us solve project planning problems through mathematical models, let's first define what a **project** is:

Project planning

A project consists of a set of m activities with their duration.
For example, activity A_i has an estimated duration of $d_i \geq 0$, for $i = 1, \dots, m$.

Some pairs of activities are subject to a **precedence constraint**: $A_i \propto A_j$ indicates that A_j can start **only after the end** of A_i.

A project can be represented by a **directed graph** G where:

- Each arc corresponds to an activity
- The arc's length represents the duration of the activity

To account for the precedence constraints, the arcs must be positioned so that if $A_i \propto A_j$, there exists a directed path where the arc associated to A_i **precedes** the arc associated to A_j.

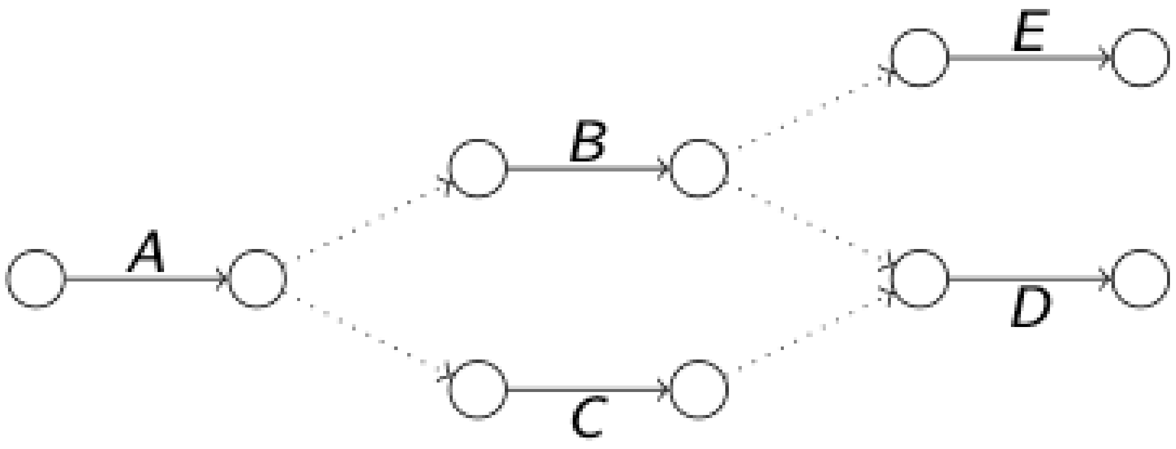
Each node v in the graph corresponds to the event "end of all the activities $(i, v) \in \delta^-(v)$" and, hence, to the possible beginning of all the activities $(v, j) \in \delta^+(v)$.

Given these definitions, we can say that the directed graph G representing any project is **acyclic**.

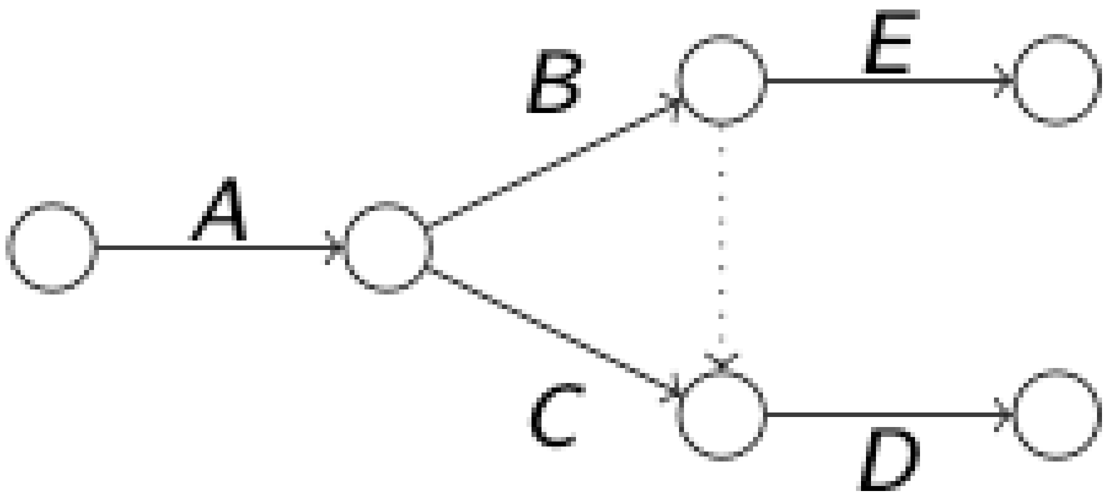
Let's look at an example: given the activities A, B, C, D, E and the following precedences:

- $A \propto B$
- $A \propto C$
- $B \propto D$
- $C \propto D$
- $B \propto E$

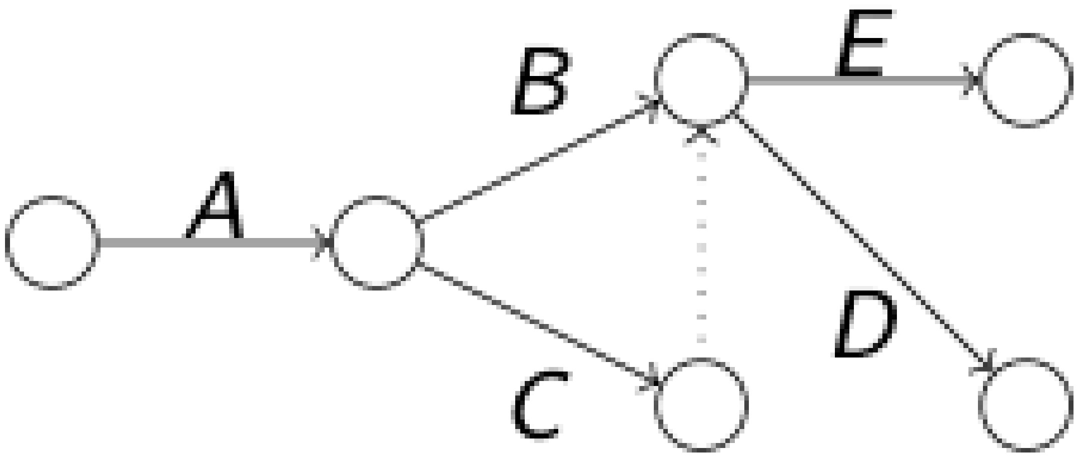
We can build the graph for this project like so:



We can represent the precedences between activities through **dummy activities** with **null duration**. To simplify the graph further, we can then contract those dummy activities in order to achieve a graph that looks like this:



It is very important **not to introduce unwanted precedence constraints** when collapsing dummy activities into nodes. For example, if we maintained the (C, B) arc in the graph:

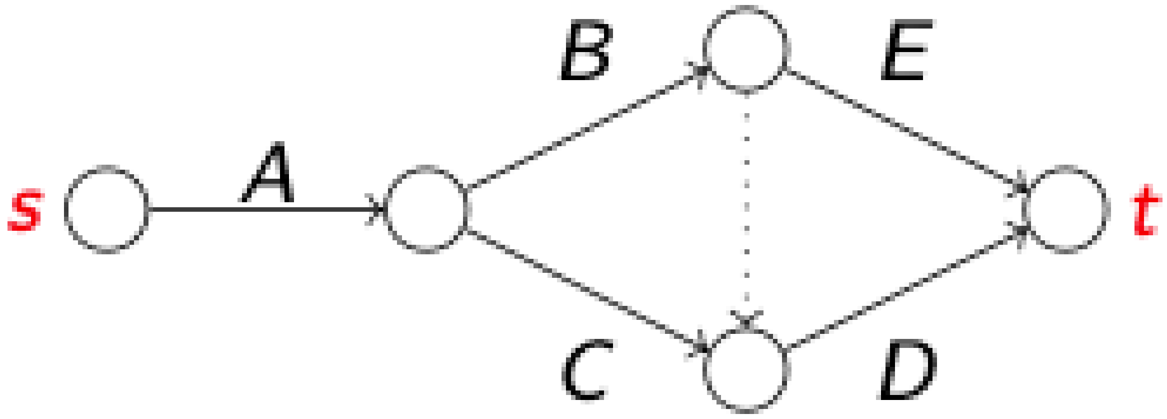


We would be implying the unwanted precedence $C \propto E$.

In order to specify the start and the end of the project, we can introduce some more simplifications:

- Every project has a unique initial node s corresponding to the beginning of the project
- Every project has a unique final node t corresponding to the end of the project
- No project contains multiple arcs (that is, arcs with the same endpoint)

Applying these simplifications brings us to this graph:



Critical Path Method

Let's try to solve the following problem:

Given a project, schedule the activities so as to minimise the overall project duration, i.e. the time needed to complete all the activities

Thanks to what we've seen up to now, we can state that the **minimum** overall project duration is the **length of a longest path** from s to t.

In order to find this longest path, we can apply the **Critical Path Method**, which determines:

- A **schedule**, specifying the order and the allotted time for the activities that minimises the overall project duration
- The **slack** of each activity, i.e. the amount of time by which its execution can be delayed without affecting the overall minimum project duration

The Critical Path Method consists of three steps:

1. Consider the nodes by increasing indices and, for each $h \in N$, find the **earliest time** $T_{\min_h}$ at which the **event** associated to node h **can occur**[3]
2. Consider the nodes by decreasing indices and, for each $h \in N$, find the **latest time** $T_{\max_h}$ at which the event associated to node h can occur **without delaying the project** completion date beyond $T_{\min_n}$
3. For each activity $(i, j) \in A$, find the **slack** $\sigma_{ij} = T_{\max_j} - T_{\min_i} - d_{ij}$

This process has a time complexity of $O(n + m)$.

If an activity in the project has zero slack, we say that it is a **critical activity**. A **critical path** is a path from s to t only composed of critical activities. In every project, a critical path can **always be found**.

2.4. Network flows

Network flows are problems involving the distribution of a given product from a set of sources to a set of users, so as to optimise a given objective function, which can be to maximise the amount of product sent through the flow (throughput), to minimise the cost of sending a given amount of product through the flow...

In order to model these problems, let's start with some definitions:

ⓘ **Network**

A **network** is a directed and connected graph $G = (N, A)$ with a source $s \in N$ and a sink $t \in N$, with $s \neq t$ and a capacity $K_{ij} \geq 0$ for each arc $(i, j) \in A$. Furthermore, the following equation holds:

$$\delta^-(s) = \delta^+(t) = \emptyset$$

In short, a network is a directed, connected graph where we have a "starting" node, the source, with no incoming arcs, and an "ending" node, the sink, with no outgoing arcs.

Each arc in the network has a **capacity**, which is the **maximum** amount of product that can pass for a given arc[4].

Let's now take a look at what a **feasible flow** is:

ⓘ **Feasible flows**

A **feasible flow** $\underline{x}$ from s to t is a **vector** $\underline{x} \in \mathbb{R}^m$ with a component x_{ij} for each arc $(i, j) \in A$ satisfying the **capacity constraints**:

$$0 \leq x_{ij} \leq k_{ij}, \ \forall (i, j) \in A$$

and the **flow balance constraints** at each intermediate node $u \in N$, with $u \neq s, t$:

$$\sum_{(i,u) \in \delta^-(u)} x_{iu} = \sum_{(u,j) \in \delta^+(u)} x_{uj}, \ \forall u \in N \setminus \{s, t\}$$

The value of the flow $\underline{x}$ is:

$$\varphi = \sum_{(s,j) \in \delta^+(s)} x_{sj}$$

Given a network and a feasible flow $\underline{x}$, an arc $(i, j) \in A$ is **saturated** if $x_{ij} = k_{ij}$, while it is **empty** if $x_{ij} = 0$.

In short, a feasible flow is a vector of elements that **specifies the amount of product passing through every arc** in a network. For this reason, every element x_{ij} of this vector must have a value that is between zero and the capacity k_{ij} of the arc it's referred to.

Furthermore, every node in the network must balance the input and output amounts: that is, if a product enters a given node, then a product must exit that same node.

We can define the **value** of the flow as the amount of product that **leaves the source node**.

Finally, if an arc in the network has a flow that is equal to its capacity, we say that arc is **saturated**; on the contrary, if any arc in a network has a null flow ($x_{ij} = 0$), we can say it's **empty**.

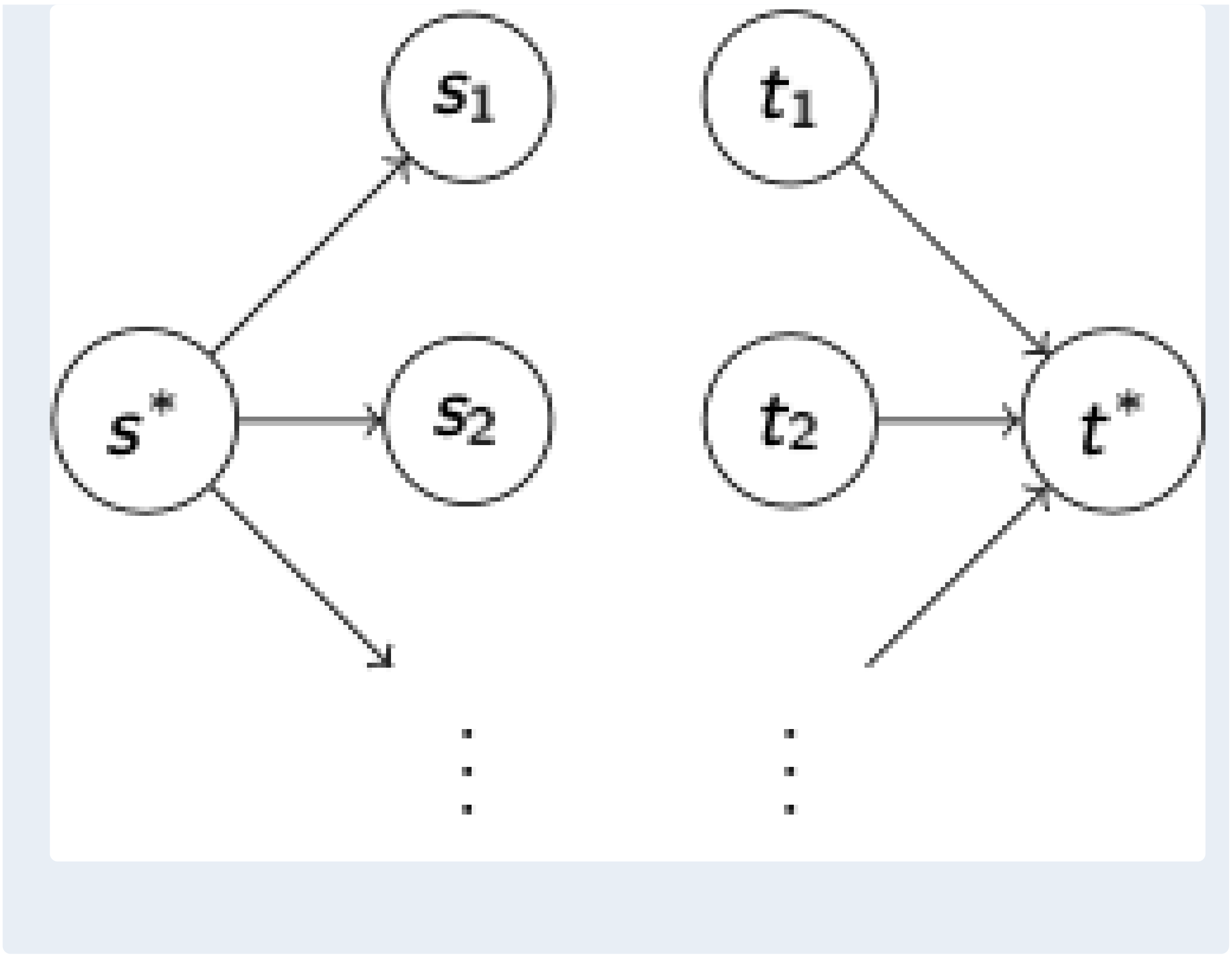
2.4.1. Maximum flow problem

Now that we've defined our basic building blocks, we can start defining some problems to solve with them:

> Given a network $G = (N, A)$ with an integer capacity k_{ij} for each arc $(i, j) \in A$ and nodes $s, t \in N$, determine a feasible flow from s to t of **maximum value**.

In practice, we want to identify the maximum amount of product that we can send from a source to a sink through a given network.

If there are many sources or sinks with a unique type of product, dummy nodes s^* and t^* can be added:



In order to solve this problem, we can define a linear programming model based on the value of the feasible flow φ:

$$\begin{aligned}
&\max \varphi \text{ so that:}\\
&\sum_{(u,j)\in\delta^+(u)} x_{uj} - \sum_{(i,u)\in\delta^-(u)} x_{iu} = \begin{cases} \varphi, & u = s \\ -\varphi, & u = t \\ 0, & \text{otherwise} \end{cases}\\
&0 \le x_{ij} \le k_{ij}, (i,j) \in A\\
&\varphi \in \mathbb{R}, x_{ij} \in \mathbb{R}, (i,j) \in A
\end{aligned}$$

Let's delve a little deeper into this problem formulation:

- The first equation maximises the amount of product sent from the source into the network
- The second line expresses the **conservation** constraint: every node (except for source and sink) must have the same amount of input and output product
- The third line expresses the **capacity** constraint of each arc: every arc must send an amount of product that is between zero and the capacity of that arc
- The last line specifies the nature of the variables and parameters used

2.4.2. Cuts, feasible flows and weak duality

We can define cuts once again in the context of flows:

Cuts and their capacity

A **cut separating s from t** is $\delta(S)$ of G with $s \in S \subset N$ and $t \in N \setminus S$. The capacity of the cut $\delta(S)$ induced by S can be calculated like this:

$$k(S) = \sum_{(i,j)\in\delta^+(S)} k_{ij}$$

Through this definition, we can find cuts that divide the source from the sink.

The number of separating cuts in any network is 2^{n-2}: any subset of nodes **excluding** source and sink.

Furthermore, we can calculate the value of a feasible flow through such a cut:

Feasible flows through cuts

Given a feasible flow $\underline{x}$ from s to t and a cut $\delta(S)$ separating s from t, the value of the feasible flow $\underline{x}$ through the cut $\delta(S)$ is:

$$\varphi(S) = \sum_{(i,j)\in\delta^+(S)} x_{ij} - \sum_{(i,j)\in\delta^-(S)} x_{ij}$$

Basically, we can get the value of a flow through a given cut by **adding** all the product that is **exiting** the cut and **subtracting** all the product that is **entering** such cut.

Let's now enumerate some of the properties of feasible flows:

Properties

1. Let $\underline{x}$ be any feasible flow from s to t. For every cut $\delta(S)$ separating s from t we have:

$$\varphi(S) = \varphi(\{s\})$$

2. For every feasible flow $\underline{x}$ from s to t and every cut $\delta(S)$ separating s from t, with $S \subseteq N$, we have:

$$\varphi(S) \leq k(S)$$

Property 1 is very intuitive: since, for every intermediate node, the amount of input and output product is the same, the value of the flow of every cut that separates the source from the sink is exactly the same as the value of the flow from the source itself.

Property 2 is also very simple: for any given cut separating source and sink, the value for any flow passing through it is at most equal to its capacity.

As a consequence of these two properties, if $\varphi(S) = k(S)$ for a subset $S \subseteq N$ with $s \in S$ and $t \notin S$, then $\underline{x}$ is a flow of **maximum value** and the cut $\delta(S)$ is of **minimum capacity**.

The property $\varphi(S) \leq k(S)$ for any feasible flow $\underline{x}$ and any cut $\delta(S)$ separating s from t expresses a **weak duality relationship** between this two problems:

Primal problem

Given $G = (N, A)$ with integer capacities on the arcs and $s, t \in N$, determine a feasible flow of **maximum value**

Dual problem

Given $G = (N, A)$ with integer arc capacities and $s, t \in N$, determine a cut separating s from t of **minimum capacity**

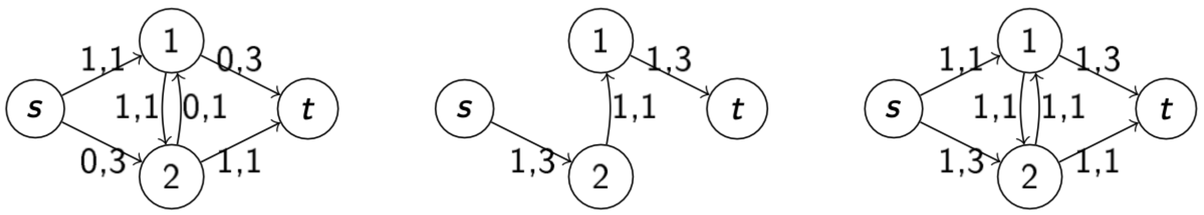
Such a relationship holds for **any Linear Programming problem**.

2.4.3. Ford-Fulkerson's algorithm

An algorithm used to solve both of the problems described above is the Ford-Fulkerson algorithm.

The idea behind it is to start from a feasible flow $\underline{x}$ and try to iteratively increase its value φ by sending, at each iteration, an additional amount of product along a path from s to t with a strictly positive residual capacity.

Let's take a look at the example below:



$\varphi_0 = \varphi(\{s\}) = 1$ $\Delta\varphi = \delta = 1$ $\varphi(\{s\}) = 2$

In the image on the left, we can see a feasible flow $\underline{x}$ of value $\varphi = 1$; furthermore, the left network has some non-saturated arcs.

We can try and increase the product throughput by sending an additional amount of product through the arc $(s, 2)$ (middle image). This would increase the value of the flow by one, so $\varphi = 2$.

The network on the right shows the final result of this iteration, with the new flow value.

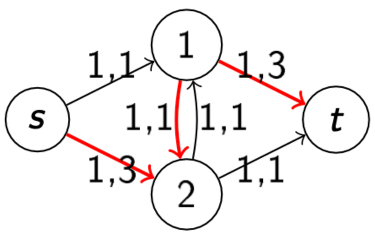
Starting from this example, can we increase the value of the flow further? Or have we reached the maximum φ value of 2?

If any arc (i, j) of the network is **not saturated**, then we can **increase** the amount of product passing through it. Furthermore, if any arc (i, j) is **not empty**, we can **decrease** the amount of product that is sent on it. This is obvious, but key to understanding Ford-Fulkerson's algorithm.

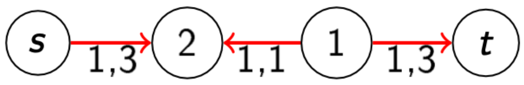
In practice, we want to try to send $\delta = 1$ **additional units of product** from s to t, which translates to:

- Sending $+\delta$ along arcs that go from s to t
- Sending $-\delta$ (technically receiving) along arcs that go from t to s

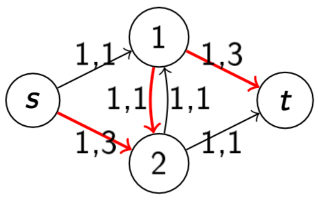
So starting from this flow:



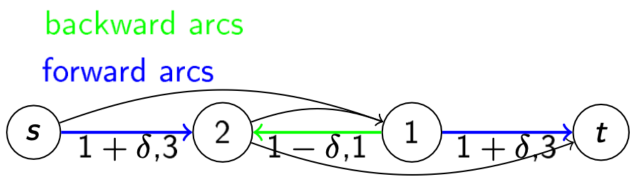
$$\varphi(\{s\}) = 2$$



We aim to get to this new one:



$$\varphi(\{s\}) = 2 + \delta$$



A path of this kind is called **augmenting path**:

Definition 5

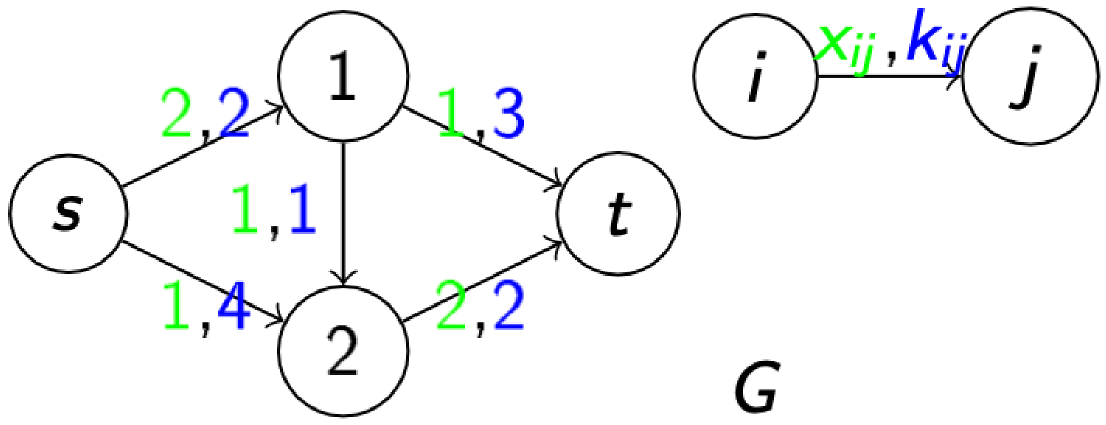
A path P from s to t is an **augmenting path** with respect to the current feasible flow $\underline{x}$ if:

- $x_{ij} < k_{ij}$ for every **forward** arc
- $x_{ij} > 0$ for every **backward** arc

We now need to find a way to look for such paths in a **systematic way**. To do so, we only need to introduce the concept of **residual network**.

A residual network $\overline{G} = (N, \overline{A})$ associated to a feasible flow $\underline{x}$ is a network that accounts for **all possible flow variations** with respect to $\underline{x}$.

For example, starting from a network G like the following:



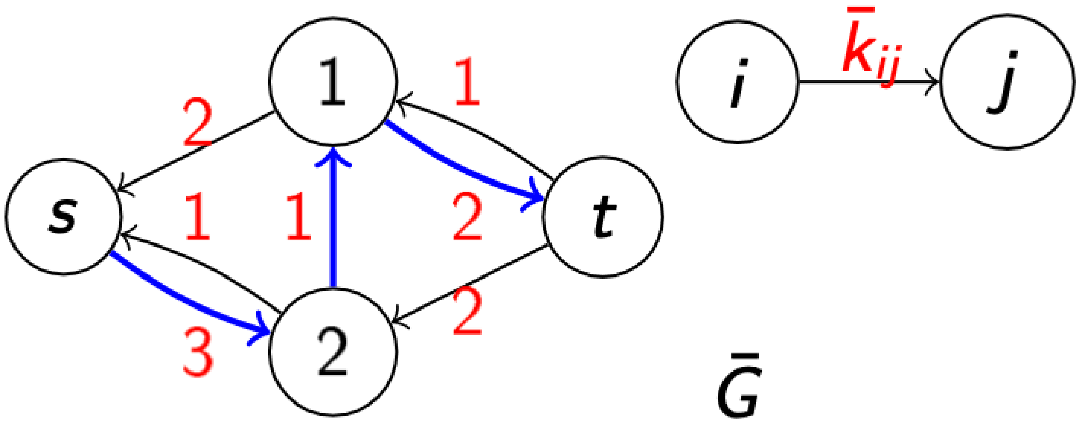
In order to build the residual network $\overline{G}$ we can follow these rules:

- If an arc (i, j) is **not empty**, then (i, j) also belongs to the residual network and has a **residual capacity** $\overline{k}_{ij} = x_{ij} > 0$
- If an arc (i, j) is **not saturated**, then (i, j) also belongs to the residual network and has a **residual capacity** $\overline{k}_{ij} = k_{ij} - x_{ij} > 0$

In practice:

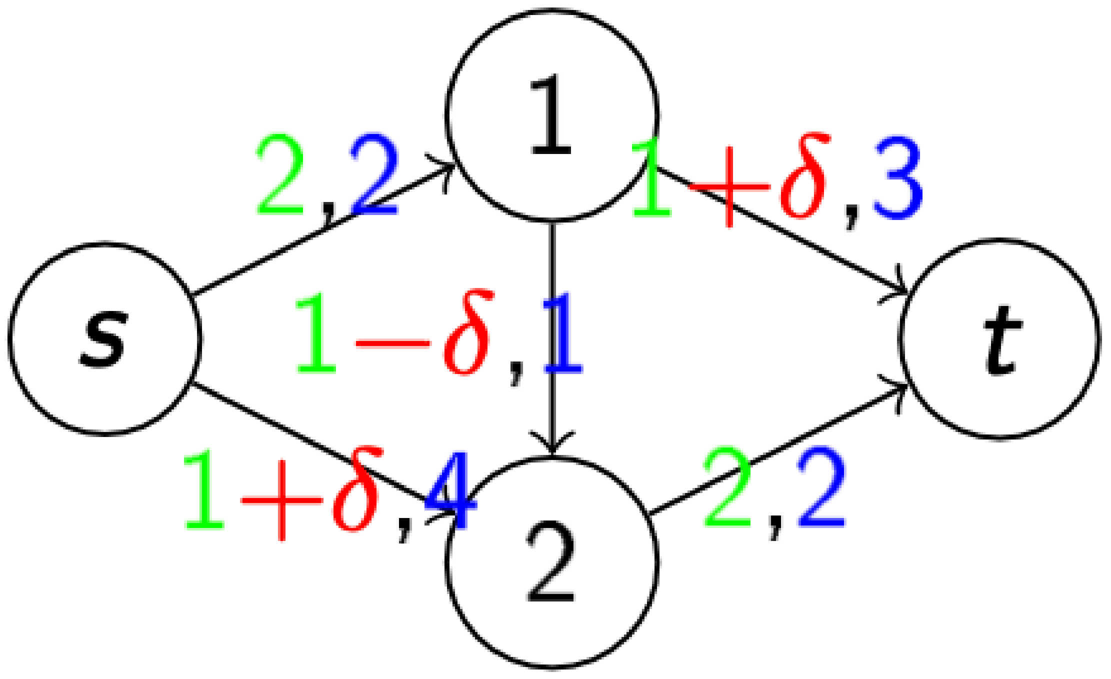
- If an arc is saturated in the original network, the only thing that we can do with that arc is to **send back** the maximum amount of product from the destination node to the source node, so we **draw a backward arc** in the residual network with a residual capacity of $\overline{k}_{ij} = k_{ij}$
- If an arc is not saturated, it can be split into two arcs:
  - A **forward arc** with a residual capacity of $\overline{k}_{ij} = k_{ij} - x_{ij}$, since we can still send some product towards the sink through that arc
  - A **backward arc** with a residual capacity of $\overline{k}_{ij} = x_{ij}$, since we can also get the sent product back through that arc

Returning to the example above, following these rules, we get the following residual network $\overline{G}$:



The advantage of using a residual network to find augmenting paths is that **there is no need to distinguish between forward and backward arcs**: any path from s to t in the residual network is an augmenting path.

Using the residual network above, we find a new augmenting path and reach this new feasible flow:



Ford-Fulkerson's algorithm works like this: at each iteration of the algorithm we look for an augmenting path in the residual network $\overline{G}$. If an augmenting path is found, then the old feasible flow is not of maximum value; otherwise, the flow $\underline{x}$ is already of maximum value and the algorithm stops.

Proposition 1

Ford-Fulkerson's algorithm is **exact**

Thanks to these results, we can also enunciate **Ford-Fulkerson's theorem**:

Theorem 6: Strong duality theorem or Ford-Fulkerson's theorem

The value of a feasible flow of maximum value is equal to the capacity of a cut of minimum capacity.

Finally, some notes:

- If all capacities k_{ij} are integer, then the flow of maximum value $\underline{x}$ has integer elements only and the value of the flow is integer as well
- Ford-Fulkerson's algorithm is not greedy, since the values of the arcs in the feasible flow can be decreased

The overall complexity of Ford-Fulkerson's algorithm is $O(m^2 k_{\max})$.

Since $k_{\max}$ can be extremely large, we can consider its size through the number of bits needed to describe it:

Definition 7

The **size of an instance** I, denoted by $|I|$, is the number of bits needed to describe the instance.

Using this definition, since $\lceil \log_2 i \rceil + 1$ bits are needed to store integer i, then we get:

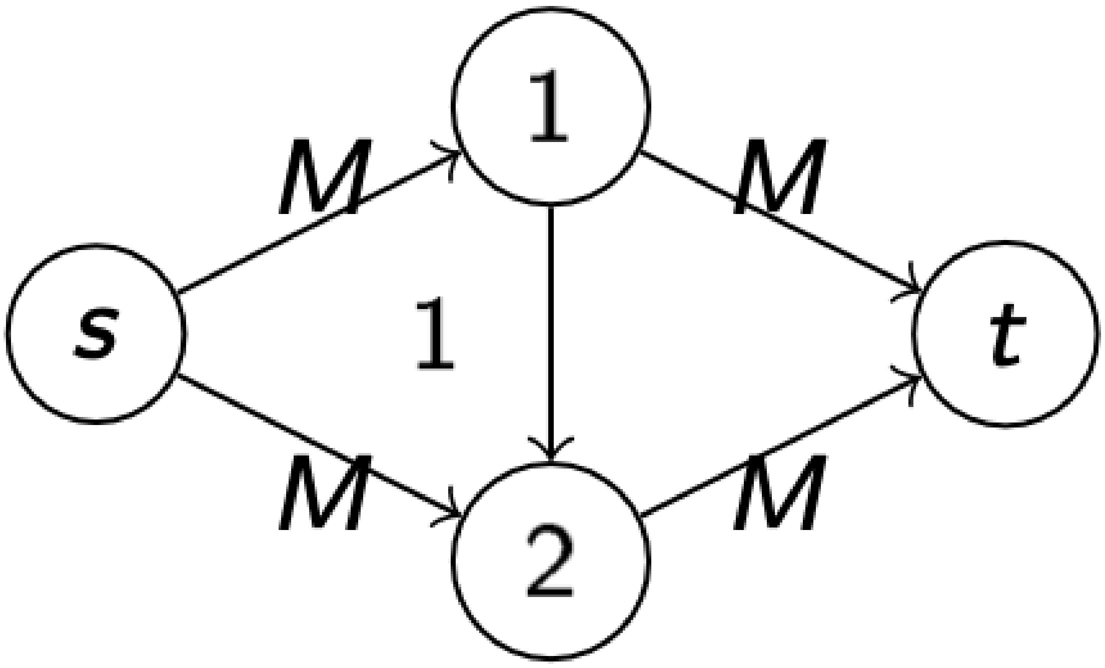
$$|I| = O(m \log_2 k_{\max})$$

Given that:

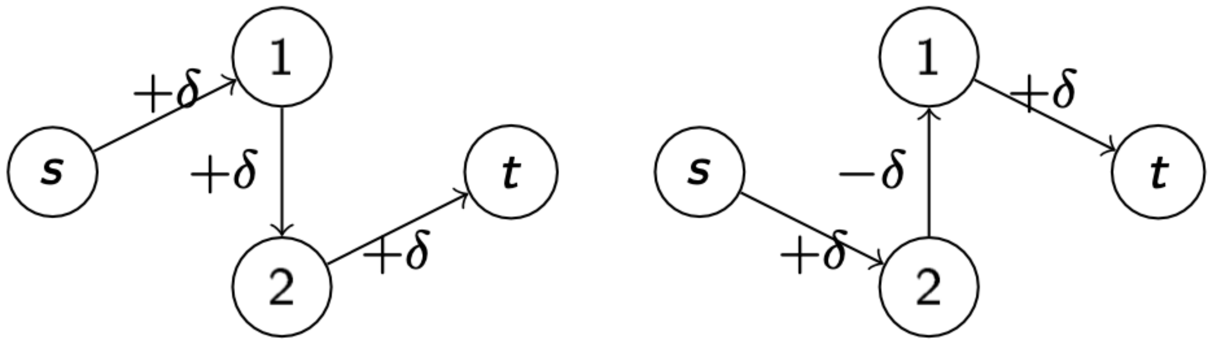
$$k_{\max} = 2^{\log_2 k_{\max}}$$

We can now say that the complexity of Ford-Fulkerson's algorithm, $O(m^2 k_{\max})$, grows **exponentially** with respect to $|I|$.

So, in some cases, this algorithm **may be very inefficient**. Let's look at an example:



Let's assume that M is a very large number. Let's build the residual network for this example:



In the worst case (that is if $\delta = 1$), we get $2M$ iterations of the algorithm.

We can make the algorithm polynomial by looking for augmenting paths with a **minimum number of arcs**. Specifically:

- Edmonds and Karp algorithm: $O(nm^2)$
- Dinic algorithm: $O(n^2 m)$

Both of these algorithms are valid even if the arc capacities are not integer.

Polynomial time algorithms for flow problems

For maximum flow problems, more efficient algorithms exists. These are based on augmenting paths, pre-flows (relaxing the node flow-balance constraints) and capacity scaling.

Problem 8: minimum cost flow problem

Given a network with a unit cost c_{ij} associated to each arc (i, j) and a value $\varphi > 0$, determine a **feasible flow** from s to t of value φ and of **minimum total cost**.

The idea behind the solution of this problem is to start from a feasible flow $\underline{x}$ of value φ and send, at each iteration, an additional amount of product in the residual network along **cycles of negative cost**.

2.5. Hard graph optimisation problems

Goal of computational complexity

The goal of computational complexity is to evaluate the computational requirements (specifically, we'll focus on time) to solve computational problems.

The two major problems related to computational complexity are:

- Evaluating the complexity of a given algorithm A to solve a given problem P
- Evaluating the inherent difficulty of a given problem P

The focus of this section is **discrete optimisation problems**.

Algorithm complexity: recap

The goal of calculating algorithm complexity is to estimate alternative algorithms for a given problem in order to select the most appropriate one for the instances of interest.

Definition 1

An **instance** I of a problem P is a special case of P.
The **size** of an instance I, denoted by $|I|$, is the number of bits needed to encode (and so describe) I.

In order to calculate the **time complexity** of an algorithm we look for a function $f(n)$ such that the number of elementary operations to solve instance I is $\leq f(n)$, for all the instances I with $|I| \leq n$.

Notes

- Since $f(n)$ is an upper bound for all instances of a problem, this is **worst-case analysis**
- $f(n)$ is expressed in **asymptotic terms** (generally through $O(\ldots)$ notation)

Definition 2

An algorithm is **polynomial** if it requires, in the worst case, a polynomial number of elementary operations:

$$f(n) = O(n^d)$$

where d is a constant and $n = |I|$.

We will usually distinguish between polynomial ($O(n^d)$) and exponential ($O(2^n)$) algorithms.

Warning

Polynomial algorithms with $d \geq 6$ are **not efficient** in practice.

Let's look at some examples:

- **Dijkstra's algorithm** for the shortest path problem: time complexity of $O(n^2)$, where n is the number of nodes. This algorithm is polynomial with respect to the size of the instance.
- The basic version of **Ford-Fulkerson's algorithm** for the maximum flow problem: time complexity of $O(m^2 k_{\max})$, where m is the number of arcs. This algorithm is not polynomial with respect to the size of the instance.

Inherent problem complexity

The goal of estimating inherent problem complexity is to evaluate the **inherent difficulty** of a given computational problem in order to adopt an appropriate solution.

In this operation, we look for the complexity of the most efficient algorithm that could ever be designed.

Definition 3

A problem P is **polynomially solvable** (or *easy*) if there is a polynomial time algorithm providing a (not necessarily optimal) solution for every instance of it.

Some examples of *easy* problems could be:

- Minimum spanning trees
- Shortest paths
- Maximum flows

For many discrete optimisation problems, the best algorithm known today requires a number of elementary operations which grows, in the worst case, exponentially with the size of the instance, but this does not prove that these problems are *difficult*.

Travelling Salesman Problem (TSP)

Problem

Given a directed graph $G = (N, A)$ with cost $c_{ij} \in \mathbb{Z}$ for each arc $(i, j) \in A$, determine a **circuit of minimum total cost** visiting **every node exactly once**.

In order to solve this problem, it is useful to define what a **Hamiltonian circuit** is:

Definition 4

A **Hamiltonian circuit** C of a graph G is a circuit that visits every node exactly once.

Denoting with H the set of all Hamiltonian circuits of a given graph G, the TSP amounts to:

$$\min_{C \in H} \sum_{(i,j) \in C} c_{ij}$$

It must be noted that H contains a finite, but exponential number of elements:

$$|H| \leq (n-1)!$$

NP-completeness theory

NP-hard computational problems are **at least as difficult** as a wide range of very challenging problems for which **no polynomial time algorithm** is known to date.

The NP-hardness of a problem is a very strong evidence that it is inherently difficult, but that does not mean that it cannot admit a polynomial time algorithm.

If any NP-hard problem admits a polynomial time algorithm, then a huge variety of very challenging computational problems that are not known to be polynomially solvable would admit a polynomial time solution.

Some examples of NP-hard problems are:

- Travelling Salesman Problem (TSP)
- Vehicle Routing Problem (VRP)

- Given a directed graph $G = (N, A)$ with arc weights, two nodes s and t, find a simple path from s to t of maximum total weight
- Integer Linear Programming (ILP)

---

1. An algorithm is **exact** if it provides an optimal solution for every instance, otherwise it is **heuristic**. ↩
2. A **greedy** algorithm constructs a feasible solution iteratively by making a "locally optimal" choice at each step, without reconsidering previous choices. ↩
3. T_{min_n} corresponds to the minimum project duration ↩
4. Sometimes, a capacity can be expressed in amount of product per unit of time ↩